Object Oriented Programming

Final Project:
A Music Quiz

Brandon Salim – 2602177783

11 June 2023

TABLE OF CONTENTS

I. PROJECT SPECIFICATION

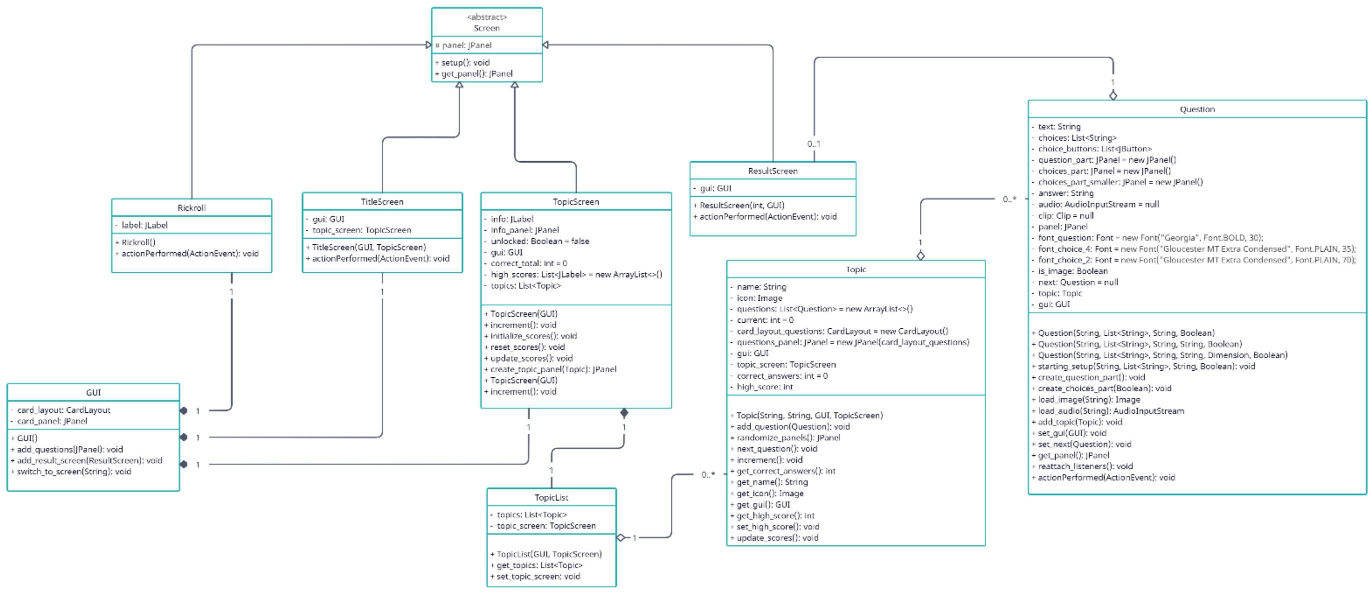
This project, in short, is about creating a music quiz. As someone who's been learning about music for as long as I can remember, I like sharing knowledge on this topic to others. I got the idea for the project when I was doing a quizziz in class, which is why I settled on this topic for my project. After settling on what the project is about, I had to determine the topics for each quiz. Here's what I decided upon.

The first topic is music theory. This is a given, and is the reason why I thought of doing this project in the first place. It's about general music theory, such as intervals, key and time signatures, chords, terms, ornaments, etc. The next topic is about general knowledge and trivia. I know that music theory will be too difficult for people who have no knowledge in that subject, so this topic is a lot easier. It's about generic questions regarding instruments, songs, composers, singers, etc. It also might have some theory questions, but the ones here are ones with obvious answers. The third topic I decided upon is music genres and video game music. I joined these two topics together to have enough questions to ask. The questions here are a mixed bag. There are some questions about genres and subgenres, perhaps about their name or origin, and some terms regarding game music. However, there are also some questions that are for jokes. For example, asking for the name of the song of an audio clip that only plays the first note of Megalovania, or asking the player to determine the genre of a song from an image, which they can only do if they know Rhythm Heaven. Regardless, this is more a fun one. I wasn't sure what to do for the last topic, so I followed Mr. Jude's idea and went with sheet music. The questions here all involve sheet music of some kind. For example, there are questions about sheet music notation, and the different clefs used in a staff. There are also a bunch of questions that involve reading sheet music, or transcribing something into sheet music.

One important thing to note is that all these questions were made by me. I couldn't really find an online source for these, so I ended up making all of these questions alone. There are, in total, 30 questions per topic. On each run through the quiz, the user will be asked 10 questions only, which will be randomized every time so that it isn't entirely repetitive. For the questions themselves, I first settled on 3 types. The first is a regular MCQ question. There's a question and 4 choices. The second is an image question. There's a question, an image that relates to the question, and 4 choices. The third is an audio question. There's a question, a button that plays an audio file, and 4 choices. Something to note here, is that the audio file will restart from the beginning if you press it while it is already playing, and there isn't a stop button. This may seem weird, but when I tried it for myself, listening randomly in the middle of the audio file, especially on some certain questions, will be problematic. The easiest example here is the question where you need to determine an irregular time signature. After I determined those types, I added a way to make each of those types a True/False question instead with only two choices. However, after I agreed on using Mr. Jude's idea for the last topic, I had to make another type of choice. So, I also added questions that used images as the choices, because images are needed to show the sheet music.

I'll quickly explain how the quiz works. The user presses start on the title screen, and gets to the topic screen. Here, they can choose one of four topics, with one secret topic locked behind 30 total correct answers. Each topic has a high score, out of 10, which they can reset in the title screen. I decided to only put the reset button there since I don't see a reason why someone would play the quiz, get a high score, and then delete their own high score immediately. By putting it in the title screen, I can see a scenario where a different person wants to play the quiz, so they reset the high scores first so that it reflects their high scores. Anyways, once you choose a topic, you will be presented by 10 questions in total, randomly selected from the 30 questions in the topic. If you answer correctly, the button will turn green, and if you answer incorrectly, the button will turn red and the correct answer will turn green. For questions that use images as choices, a green and red border will appear instead. Once the user answers all the questions, they will get a result screen that shows the number of correct answers they got on that run, and a button to return to the topic selection screen. If that run is a high score, the user will now see that high score under the topic. The number of total correct answers will update, until it reaches 30, which unlocks the secret topic. The secret topic leads to a screen with text and a button, which plays an mp4 of Never Gonna Give You up, locking you in that screen where the only option is closing the program. Another thing is that after you answer a question, there will be a 4 second delay before you move to the next question. This should give an opportunity for the user to see whether or not they got it right, and look at the right answer. I used a button at first, but after testing, I liked the 4s timer better because it's just more convenient.

II. Solution Design



<abstract>
Screen
# panel: JPanel
+ setup(): void
+ get_panel(): JPanel
Rickroll
- label: JLabel
+ Rickroll()
+ actionPerformed(ActionEvent): void
TitleScreen
- gui: GUI
- topic_screen: TopicScreen
+ TitleScreen(GUI, TopicScreen)
+ actionPerformed(ActionEvent): void
TopicScreen
- info: JLabel
- info_panel: JPanel
- unlocked: Boolean = false
- gui: GUI
- correct_total: int = 0
- high_scores: List<JLabel> = new ArrayList<>()
- topics: List<Topic>
+ TopicScreen(GUI)
+ increment(): void
+ initialize_scores(): void
+ reset_scores(): void
+ update_scores(): void
+ create_topic_panel(Topic): JPanel
+ TopicScreen(GUI)
+ increment(): void
ResultScreen
- gui: GUI
+ ResultScreen(int, GUI)
+ actionPerformed(ActionEvent): void
GUI
- card_layout: CardLayout
- card_panel: JPanel
+ GUI()
+ add_questions(JPanel): void
+ add_result_screen(ResultScreen): void
+ switch_to_screen(String): void
TopicList
- topics: List<Topic>
- topic_screen: TopicScreen
+ TopicList(GUI, TopicScreen)
+ get_topics: List<Topic>
+ set_topic_screen: void
Topic
- name: String
- icon: Image
- questions: List<Question> = new ArrayList<>()
- current: int = 0
- card_layout_questions: CardLayout = new CardLayout()
- questions_panel: JPanel = new JPanel(card_layout_questions)
- gui: GUI
- topic_screen: TopicScreen
- correct_answers: int = 0
- high_score: int
+ Topic(String, String, GUI, TopicScreen)
+ add_question(Question): void
+ randomize_panels(): JPanel
+ next_question(): void
+ increment(): void
+ get_correct_answers(): int
+ get_name(): String
+ get_icon(): Image
+ get_gui(): GUI
+ get_high_score(): int
+ set_high_score(): void
+ update_scores(): void
Question
- text: String
- choices: List<String>
- choice_buttons: List<JButton>
- question_part: JPanel = new JPanel()
- choices_part: JPanel = new JPanel()
- choices_part_smaller: JPanel = new JPanel()
- answer: String
- audio: AudioInputStream = null
- clip: Clip = null
- panel: JPanel
- font_question: Font = new Font("Georgia", Font.BOLD, 30);
- font_choice_4: Font = new Font("Gloucester MT Extra Condensed", Font.PLAIN, 35);
- font_choice_2: Font = new Font("Gloucester MT Extra Condensed", Font.PLAIN, 70);
- is_image: Boolean
- next: Question = null
- topic: Topic
- gui: GUI
+ Question(String, List<String>, String, Boolean)
+ Question(String, List<String>, String, String, Boolean)
+ Question(String, List<String>, String, String, Dimension, Boolean)
+ starting_setup(String, List<String>, String, Boolean): void
+ create_question_part(): void
+ create_choices_part(Boolean): void
+ load_image(String): Image
+ load_audio(String): AudioInputStream
+ add_topic(Topic): void
+ set_gui(GUI): void
+ set_next(Question): void
+ get_panel(): JPanel
+ reattach_listeners(): void
+ actionPerformed(ActionEvent): void
1
0..1
0..*

Here is my UML Class Diagram. I will list down each class, and explain what they do.

I. Screen is an abstract class. It's very simple, and is meant to create a panel and set up its alignment and background, which all the classes that uses panels share. It also has a method to return the panel.

II. The GUI class is the class that contains the JFrame, and is the driver. It has the main card layout for the entire program, and calls 1 instance of the Rickroll class, 1 instance of the TitleScreen class and 1 instance of the TopicScreen class in its constructor.

III. The Rickroll class is a class for the panel with the text and the button, where the button plays the mp4. It extends the Screen abstract class.

IV. The TitleScreen class is a class for the title screen panel. It has text and two buttons, one to start and one to reset scores. It extends the Screen abstract class.

V. The TopicScreen class is a class for the topic screen panel. It shows all the topics, their high scores, the secret topic, and the instructions to unlock it. It creates 1 instance of the TopicList class in its constructor, which is used to populate the gridlayout panel with topics. It extends the Screen abstract class.

VI. The TopicList class is a class that contains a list of all the topics in the quiz, where each topic is an instance of the Topic class.

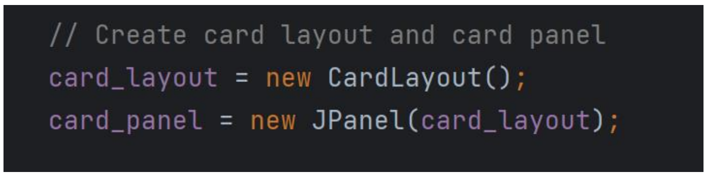
VII. The Topic class is a class for a single topic. It has its name, the image to show in the topic screen, all of its questions (where each is an instance of the Question class), a card layout for the questions to be shown, the high score for that topic, the number of correct answers in a run, and pointers. It is also responsible for randomizing the questions and switching to the next question when the user plays a quiz.

VIII. The Question class is a class for a single question, and the panel for that question. It has 3 different constructors, one for a text question, one for an image question, and one for an audio question. Each constructor creates a panel for the question, based on its type, the number of choices, and the type of choices. The buttons here are also responsible for calling the method in Topic to switch to the next question. The last Question object in the list for a single run through a quiz will also create one instance of the ResultScreen class based on the number of correct answers.

IX. The ResultSCreen class is a class for the result screen panel. It tells you how many answers you got correct, and has a single button that brings you back to the TopicScreen. It extends the Screen Abstract class

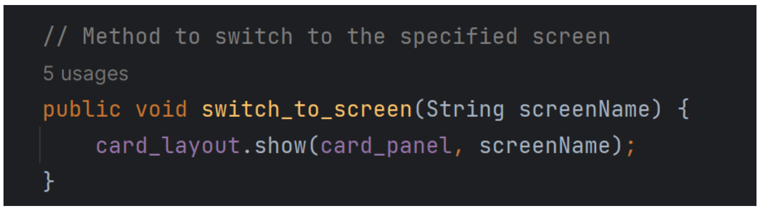
III. Explanation

Let’s start with the GUI class.



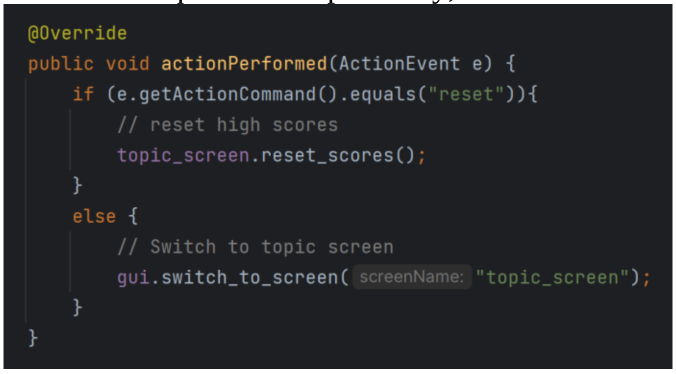
```
// Create card layout and card panel
card_layout = new CardLayout();
card_panel = new JPanel(card_layout);
```

The main panel is a card layout, which I used to be able to switch between screens. There are multiple times that the screen switch is needed in other classes too, so one of the most important (yet simple) method is just this.



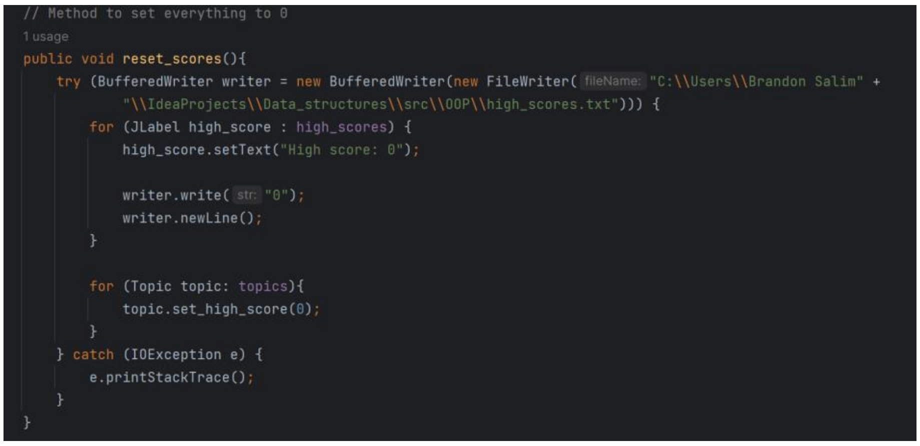
```
// Method to switch to the specified screen
5 usages
public void switch_to_screen(String screenName) {
    card_layout.show(card_panel, screenName);
}
```

Now, the title screen.
This class is quite self-explanatory, but this will be important for the next class.



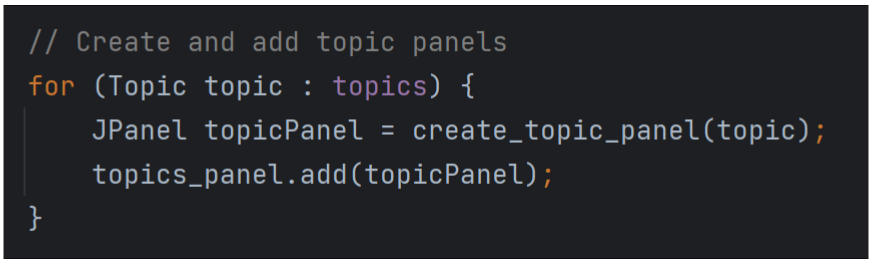
```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("reset")){
        // reset high scores
        topic_screen.reset_scores();
    }
    else {
        // Switch to topic screen
        gui.switch_to_screen( screenName: "topic_screen");
    }
}
```

First, you can see that it calls the method from the GUI class to switch to the topic screen. However, if the action command of the button is to reset, it will call the reset_scores method from the topic screen class. Let's take a look at that class next.



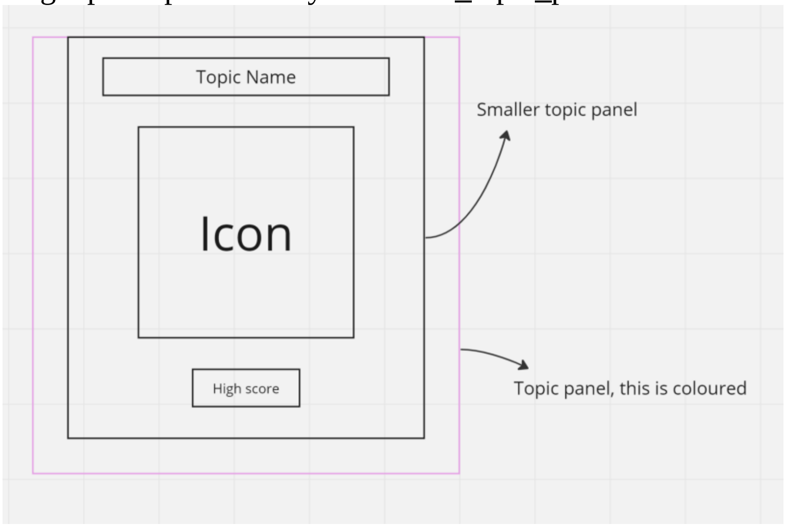
```
// Method to set everything to 0
1 usage
public void reset_scores(){
    try (BufferedWriter writer = new BufferedWriter(new FileWriter( fileName: "C:\\Users\\Brandon Salim" +
            "\\IdeaProjects\\Data_structures\\src\\OOP\\high_scores.txt"))) {
        for (JLabel high_score : high_scores) {
            high_score.setText("High score: 0");

            writer.write( str: "0");
            writer.newLine();
        }

        for (Topic topic: topics){
            topic.set_high_score(0);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

This method is used to set all high scores to 0. It first sets all the text in the topic screen to have a high score for 0, then writes 0 in every line for the text file, then set every topic's high scores as 0. There are two more similar methods to this, one to initialize scores and one to update scores. The initialize_scores method is called when the program is run, which reads lines from the text file then sets up the screen text and sets the high scores to the topics accordingly. The update_scores method is called after the high score of a topic is broken, which sets the text, sets the high score of the topic and writes it down in the text file.

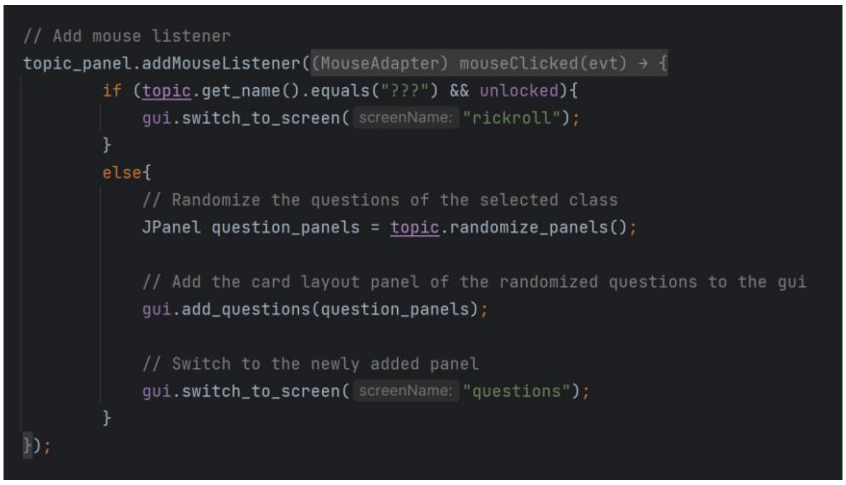


```
// Create and add topic panels
for (Topic topic : topics) {
    JPanel topicPanel = create_topic_panel(topic);
    topics_panel.add(topicPanel);
}
```

I think that the most important method in this class is the one that creates panels based on each topic is the topic list, which is then put on the main topic screen panel. Showing the whole thing is a bit long and some parts are just initializing, so I'm just going to show the layout of a single panel produced by the create_topic_panel method.



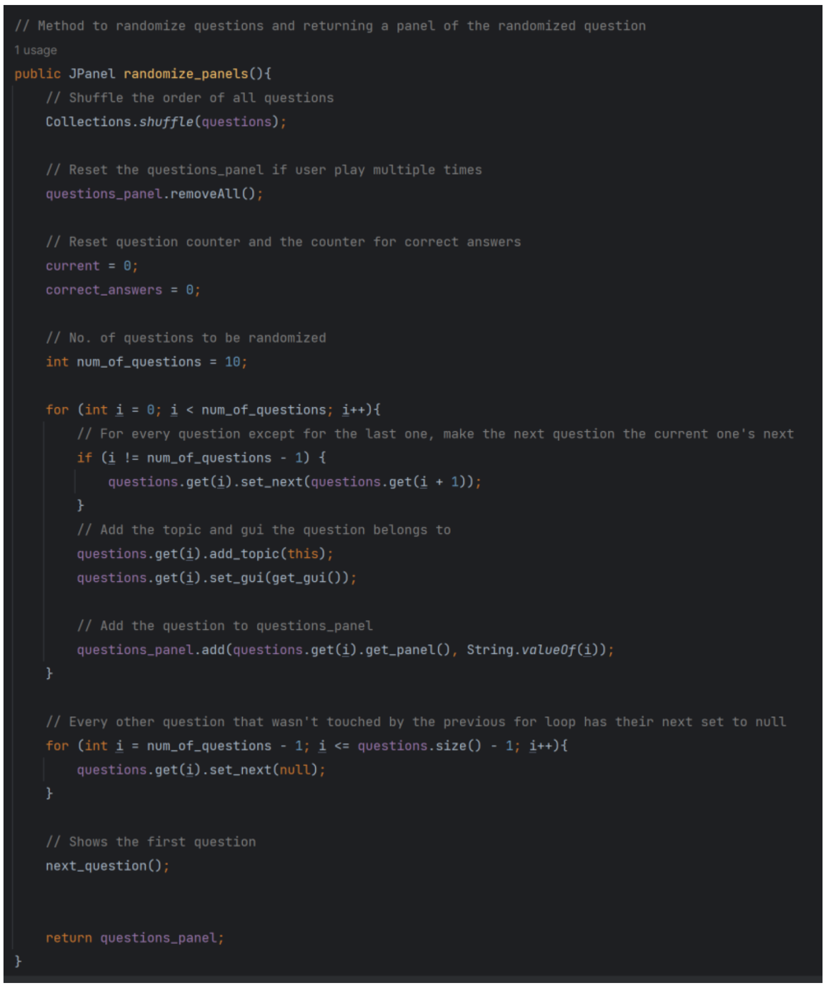
Here is a single panel for a single topic. These panels are then put into a panel with the gridlayout, with 2 topic panels per row and a dynamic number of rows.



```
// Add mouse listener
topic_panel.addMouseListener((MouseAdapter) mouseClicked(evt) → {
        if (topic.get_name().equals("???") && unlocked){
            gui.switch_to_screen( screenName: "rickroll");
        }
        else{
            // Randomize the questions of the selected class
            JPanel question_panels = topic.randomize_panels();

            // Add the card layout panel of the randomized questions to the gui
            gui.add_questions(question_panels);

            // Switch to the newly added panel
            gui.switch_to_screen( screenName: "questions");
        }
});
```

Each topic panel also has a mouse listener. The first if statement checks if the topic being pressed is the secret one, and it will only move to the rickroll screen is the Boolean unlocked is true (which only happens after you get a total of 30 correct answers). Else, it will call the randomize_panels method from the Topic class. It stores the card layout panel from that method to question_panels, which is then added to the gui, and then it switches the main panel from the GUI class to the questions panel.

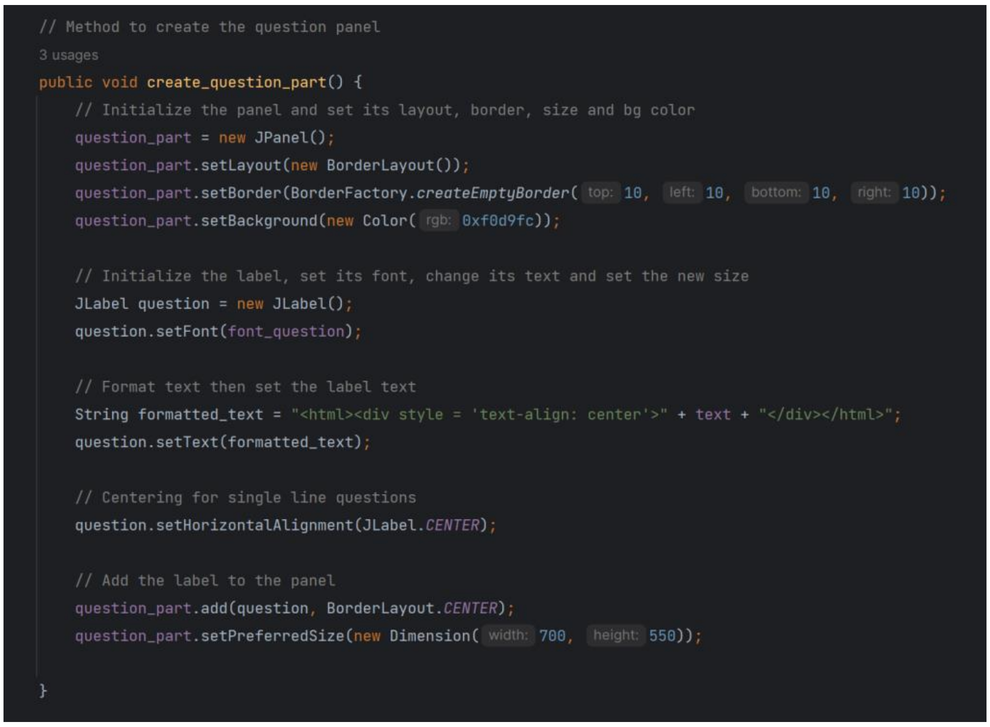
The TopicList class is quite self-explanatory, so I'll show the Topic class next. (Image on next page, it can't fit here.)



```java
// Method to randomize questions and returning a panel of the randomized question
1 usage
public JPanel randomize_panels(){
    // Shuffle the order of all questions
    Collections.shuffle(questions);

    // Reset the questions_panel if user play multiple times
    questions_panel.removeAll();

    // Reset question counter and the counter for correct answers
    current = 0;
    correct_answers = 0;

    // No. of questions to be randomized
    int num_of_questions = 10;

    for (int i = 0; i < num_of_questions; i++){
        // For every question except for the last one, make the next question the current one's next
        if (i != num_of_questions - 1) {
            questions.get(i).set_next(questions.get(i + 1));
        }
        // Add the topic and gui the question belongs to
        questions.get(i).add_topic(this);
        questions.get(i).set_gui(get_gui());

        // Add the question to questions_panel
        questions_panel.add(questions.get(i).get_panel(), String.valueOf(i));
    }

    // Every other question that wasn't touched by the previous for loop has their next set to null
    for (int i = num_of_questions - 1; i <= questions.size() - 1; i++){
        questions.get(i).set_next(null);
    }

    // Shows the first question
    next_question();


    return questions_panel;
}
```

To start, I shuffle the contents of questions, which is a list of Question objects. I reset the card layout panel for the questions by removing all previous questions, and reset the value of current and correct answers. Current is to move to the next question in another method, while correct answers is self-explanatory. Then, since I'm only taking 10 questions per play, I set the number of questions to 10. Since the list is already randomized, all I need to do then is to take the first 10 entries in the list. The if statement in the first for loop sets the next attribute in its question. This is important because the result screen will only be generated when the next attribute of a question is null. I then add the topic object and the gui object to the question, which I need to access methods from those classes in the Question class. I then add it to the question panel, and use the value of i as its identifier. I then have to set every other question's next to be null. This is to reset the next attributes of questions randomly picked in a previous run. Next_question then calls the first question, and the method returns the entire panel to the topic panel class, which was mentioned above.

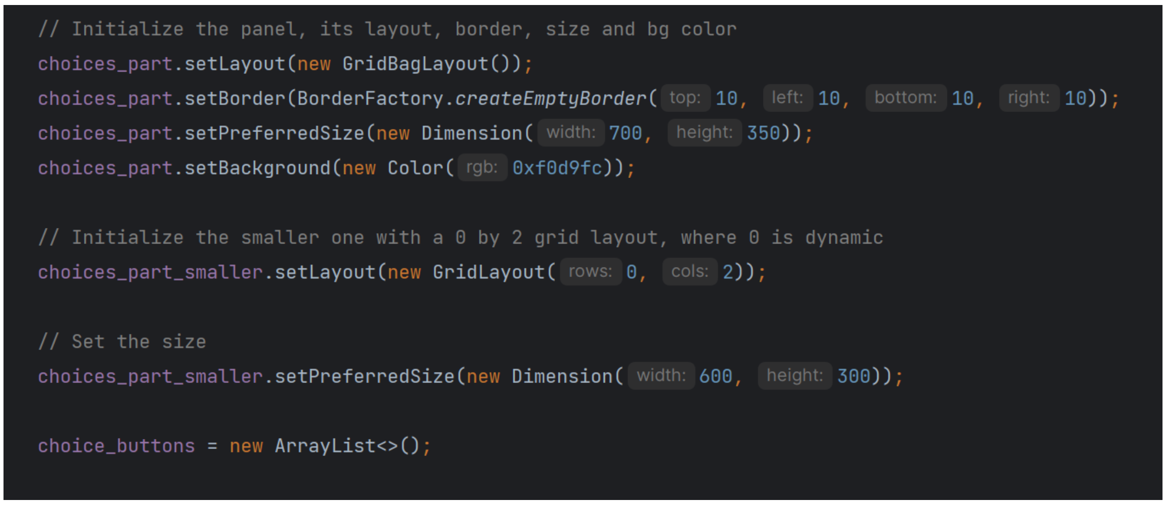
Now, let's take a look at the Question class. This will be the longest one by a long shot. To start, I designed the panel for a question to be split into two parts. The top part is for the question. It doesn't matter if the question is just text, or it has an image, or has an audio. This top part will contain the question. The bottom part is for the choices. There can be 2 or 4 choices, and the choices may also be images, but they are always in this bottom part. To reuse code, I made methods to construct these two separate parts. Let's start with the top part, since it's far simpler.



```
// Method to create the question panel
3 usages
public void create_question_part() {
    // Initialize the panel and set its layout, border, size and bg color
    question_part = new JPanel();
    question_part.setLayout(new BorderLayout());
    question_part.setBorder(BorderFactory.createEmptyBorder( top: 10, left: 10, bottom: 10, right: 10));
    question_part.setBackground(new Color( rgb: 0xf0d9fc));

    // Initialize the label, set its font, change its text and set the new size
    JLabel question = new JLabel();
    question.setFont(font_question);

    // Format text then set the label text
    String formatted_text = "<html><div style = 'text-align: center'>" + text + "</div></html>";
    question.setText(formatted_text);

    // Centering for single line questions
    question.setHorizontalAlignment(JLabel.CENTER);

    // Add the label to the panel
    question_part.add(question, BorderLayout.CENTER);
    question_part.setPreferredSize(new Dimension( width: 700, height: 550));

}
```

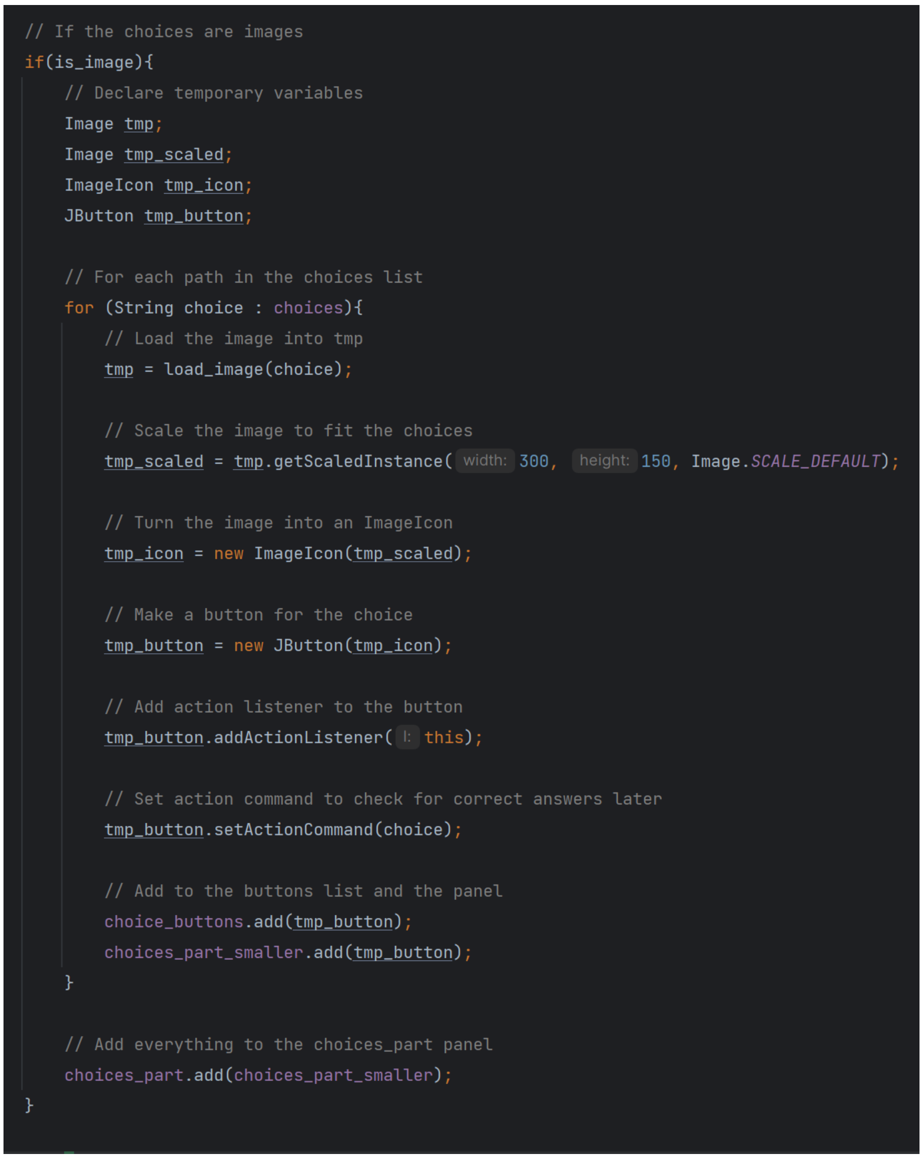
This one's quite simple. It creates a panel for this top part, then sets its layout, a small empty border around it, and the background colour. Then, it makes a label for the question and set the text to the question. The question is formatted first because most questions don't fit on the screen in one line, and the formatting wraps it in the screen and centres it. The next line centres the label horizontally, since single line questions weren't centred by the previous formatting. Finally, it puts the label on the panel and sets its size.

Now the choices part, on the other hand, is not this simple. I'll split the screenshots.



```
// Initialize the panel, its layout, border, size and bg color
choices_part.setLayout(new GridBagLayout());
choices_part.setBorder(BorderFactory.createEmptyBorder( top: 10, left: 10, bottom: 10, right: 10));
choices_part.setPreferredSize(new Dimension( width: 700, height: 350));
choices_part.setBackground(new Color( rgb: 0xf0d9fc));

// Initialize the smaller one with a 0 by 2 grid layout, where 0 is dynamic
choices_part_smaller.setLayout(new GridLayout( rows: 0, cols: 2));

// Set the size
choices_part_smaller.setPreferredSize(new Dimension( width: 600, height: 300));

choice_buttons = new ArrayList<>();
```

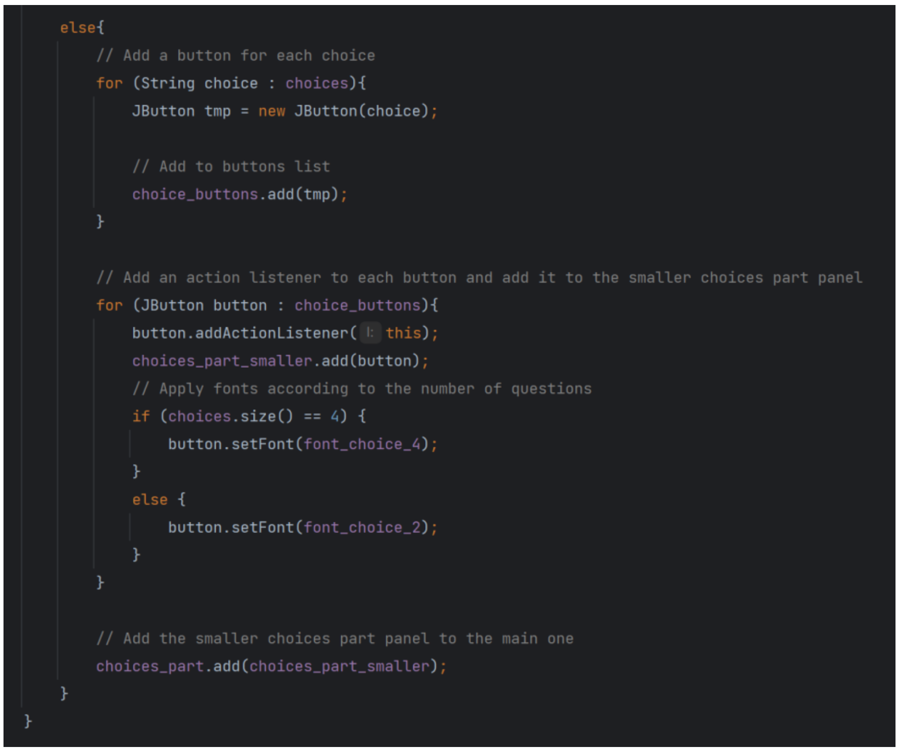
This first section is to create a panel for the choices part, and a smaller panel. The main panel uses the GridBagLayout to center its contents, and I use a grid layout for the smaller part, which will actually contain the choices. This smaller part will be added to the main choices panel in the end. I set the preferred size of both panels, and create a new array list for the choice_buttons attribute. This list will store the JButtons for each choice. This is needed to reference them later on, which we’ll get to.



```java
// If the choices are images
if(is_image){
    // Declare temporary variables
    Image tmp;
    Image tmp_scaled;
    ImageIcon tmp_icon;
    JButton tmp_button;

    // For each path in the choices list
    for (String choice : choices){
        // Load the image into tmp
        tmp = load_image(choice);

        // Scale the image to fit the choices
        tmp_scaled = tmp.getScaledInstance( width: 300,  height: 150, Image.SCALE_DEFAULT);

        // Turn the image into an ImageIcon
        tmp_icon = new ImageIcon(tmp_scaled);

        // Make a button for the choice
        tmp_button = new JButton(tmp_icon);

        // Add action listener to the button
        tmp_button.addActionListener( l: this);

        // Set action command to check for correct answers later
        tmp_button.setActionCommand(choice);

        // Add to the buttons list and the panel
        choice_buttons.add(tmp_button);
        choices_part_smaller.add(tmp_button);
    }

    // Add everything to the choices_part panel
    choices_part.add(choices_part_smaller);
}
```

This section is for questions that use images for their choices. I declare four temporary variables, then make a for loop for each choice in the choices list. If the question uses images as choices, the choices list is a list of image paths. So, I call the load_image method on the path to get its image. It’s just a simple method to read the path, but I separated it because it’s also used in the image questions. Since these are buttons, I want to make sure that all of them have the exact same size. So, tmp_scaled is assigned a scaled version of the image, and tmp_icon is an image icon of the scaled image. Then, I create a new button for each image icon, add an action listener and set the action command to the image’s path. I will use this action command to check for the correct answer later. Then I add each button to the choice_buttons list, and add it to the panel.

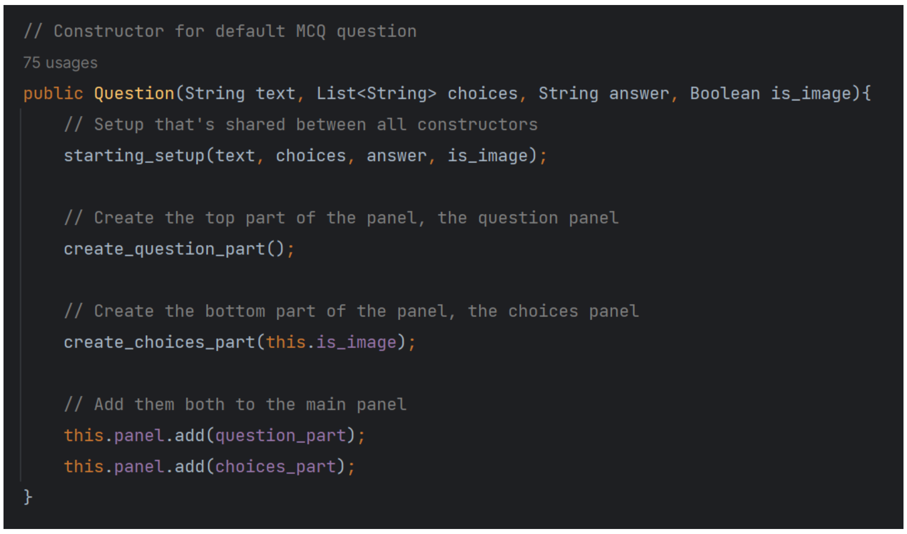
Next section is on the next page, can’t fit it here.



```java
    else{
        // Add a button for each choice
        for (String choice : choices){
            JButton tmp = new JButton(choice);

            // Add to buttons list
            choice_buttons.add(tmp);
        }

        // Add an action listener to each button and add it to the smaller choices part panel
        for (JButton button : choice_buttons){
            button.addActionListener( l: this);
            choices_part_smaller.add(button);
            // Apply fonts according to the number of questions
            if (choices.size() == 4) {
                button.setFont(font_choice_4);
            }
            else {
                button.setFont(font_choice_2);
            }
        }

        // Add the smaller choices part panel to the main one
        choices_part.add(choices_part_smaller);
    }
}
```

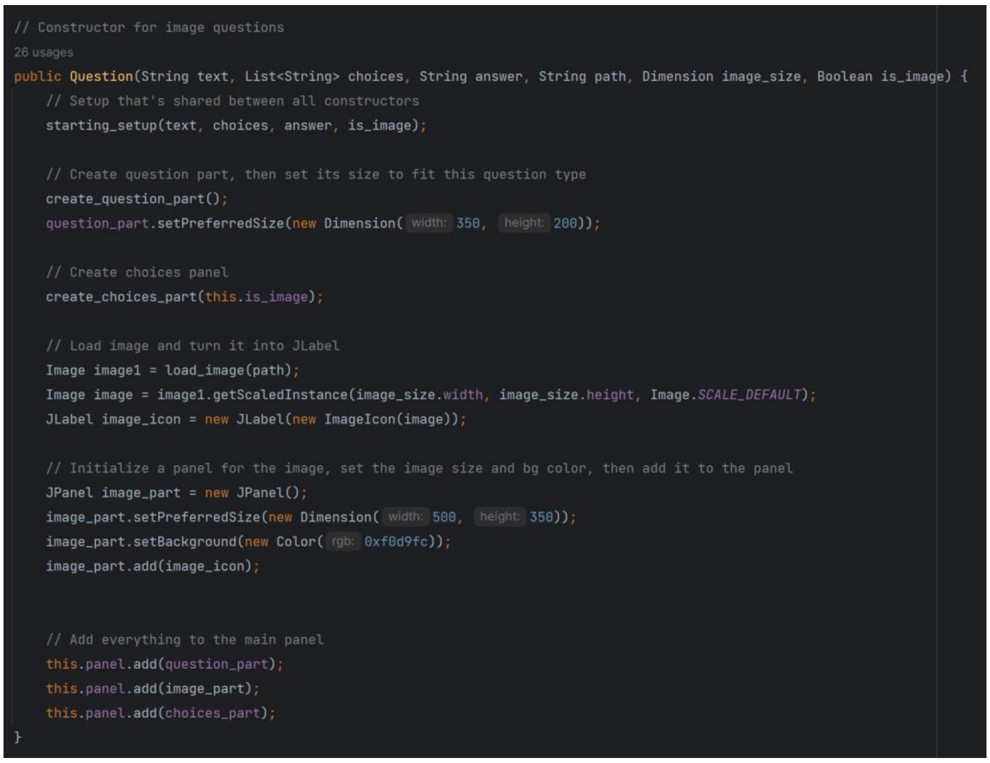
This is if the choices are text. Skipping the part where I need to take images, scale them and turn them into icons, I immediately make a JButton for each choice, setting their text with the constructor. Then, I add the action listener to each button, and add the button to the panel. If there are 4 choices, I set the font to font_choice_4, which is a bit smaller than font_choice_2. The font_choice_2 is for questions with 2 choices, which are True/False questions.

Now that these two methods are covered, let's look at the 3 constructors for this class.



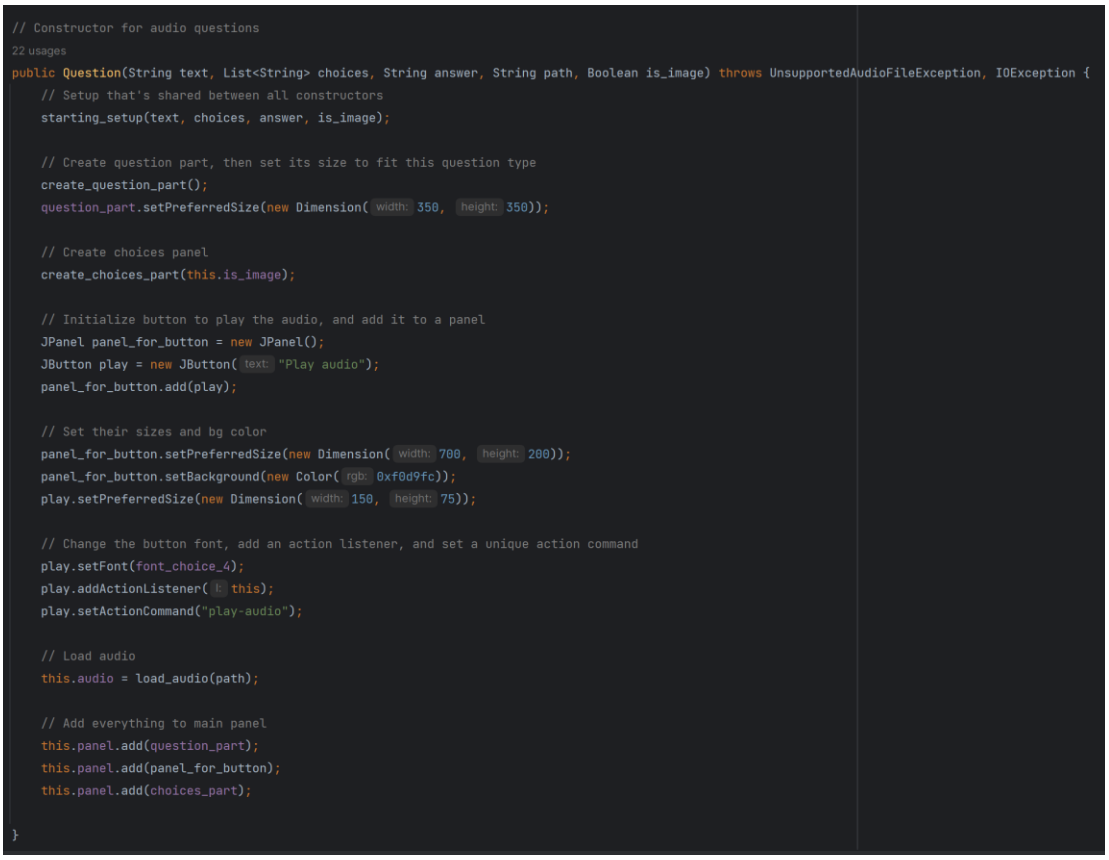
```java
// Constructor for default MCQ question
75 usages
public Question(String text, List<String> choices, String answer, Boolean is_image){
    // Setup that's shared between all constructors
    starting_setup(text, choices, answer, is_image);

    // Create the top part of the panel, the question panel
    create_question_part();

    // Create the bottom part of the panel, the choices panel
    create_choices_part(this.is_image);

    // Add them both to the main panel
    this.panel.add(question_part);
    this.panel.add(choices_part);
}
```

This is the constructor for a normal text question. This is the most basic one, so I simply call the methods and it's done. I didn't show the starting_setup earlier, but it's really simple. All it does is assign the values from the arguments in the constructor to the attributes of the class, and create the main panel and set its layout. Every constructor will need to do this setup, so I decided to turn it into a method to reduce the number of lines.



```
// Constructor for image questions
26 usages
public Question(String text, List<String> choices, String answer, String path, Dimension image_size, Boolean is_image) {
    // Setup that's shared between all constructors
    starting_setup(text, choices, answer, is_image);

    // Create question part, then set its size to fit this question type
    create_question_part();
    question_part.setPreferredSize(new Dimension( width: 350,  height: 200));

    // Create choices panel
    create_choices_part(this.is_image);

    // Load image and turn it into JLabel
    Image image1 = load_image(path);
    Image image = image1.getScaledInstance(image_size.width, image_size.height, Image.SCALE_DEFAULT);
    JLabel image_icon = new JLabel(new ImageIcon(image));

    // Initialize a panel for the image, set the image size and bg color, then add it to the panel
    JPanel image_part = new JPanel();
    image_part.setPreferredSize(new Dimension( width: 500,  height: 350));
    image_part.setBackground(new Color( rgb: 0xf0d9fc));
    image_part.add(image_icon);


    // Add everything to the main panel
    this.panel.add(question_part);
    this.panel.add(image_part);
    this.panel.add(choices_part);
}
```

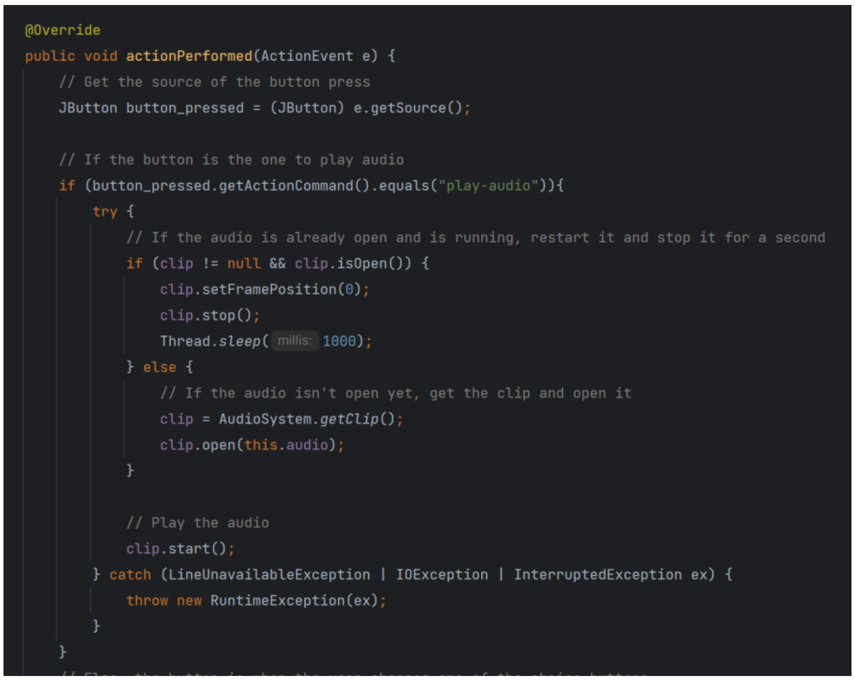
Here is the constructor for image questions. Starts the same way, but after creating the question panel, I reduce its size. I load the image with the load_image method, then scale its size to the dimensions specified in the constructor's arguments. The scaled image is put into an image icon, which is put into a JLabel. This JLabel is put into a JPanel for the image, then all three panels are put into the main panel.



```
// Constructor for audio questions
22 usages
public Question(String text, List<String> choices, String answer, String path, Boolean is_image) throws UnsupportedAudioFileException, IOException {
    // Setup that's shared between all constructors
    starting_setup(text, choices, answer, is_image);

    // Create question part, then set its size to fit this question type
    create_question_part();
    question_part.setPreferredSize(new Dimension( width: 350,  height: 350));

    // Create choices panel
    create_choices_part(this.is_image);

    // Initialize button to play the audio, and add it to a panel
    JPanel panel_for_button = new JPanel();
    JButton play = new JButton( text: "Play audio");
    panel_for_button.add(play);

    // Set their sizes and bg color
    panel_for_button.setPreferredSize(new Dimension( width: 700,  height: 200));
    panel_for_button.setBackground(new Color( rgb: 0xf0d9fc));
    play.setPreferredSize(new Dimension( width: 150,  height: 75));

    // Change the button font, add an action listener, and set a unique action command
    play.setFont(font_choice_4);
    play.addActionListener( l: this);
    play.setActionCommand("play-audio");

    // Load audio
    this.audio = load_audio(path);

    // Add everything to main panel
    this.panel.add(question_part);
    this.panel.add(panel_for_button);
    this.panel.add(choices_part);

}
```

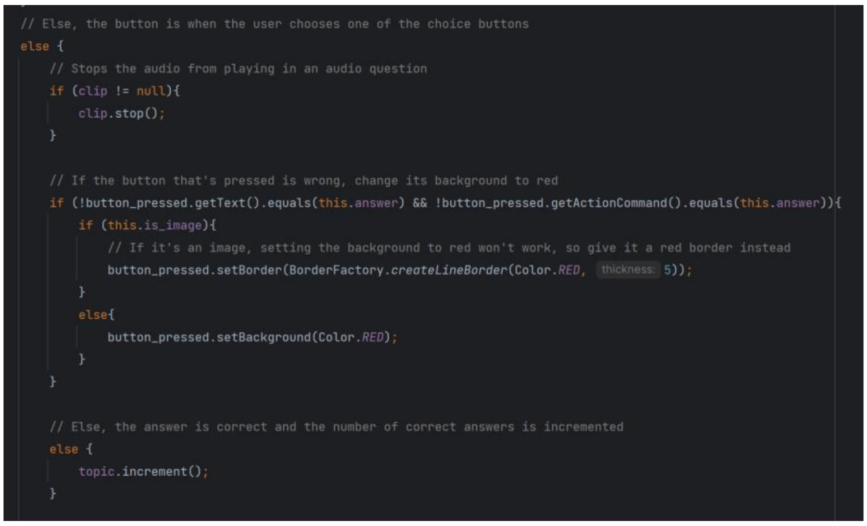
Finally, the constructor for audio questions. Starts the same way and reduces the size of the question panel, then creates a panel for the JButton, which will be used to play the audio. An action listener is added to the button after some setup, and an action command is set. This action command is needed to distinguish the button to play the audio from the choice buttons. The audio is loaded from its path into the audio attribute using the load_audio method, which returns an AudioInputStream type from the path. All three panels are then added to the main panel.

The last thing I want to show for this class is the actionPerformed method.



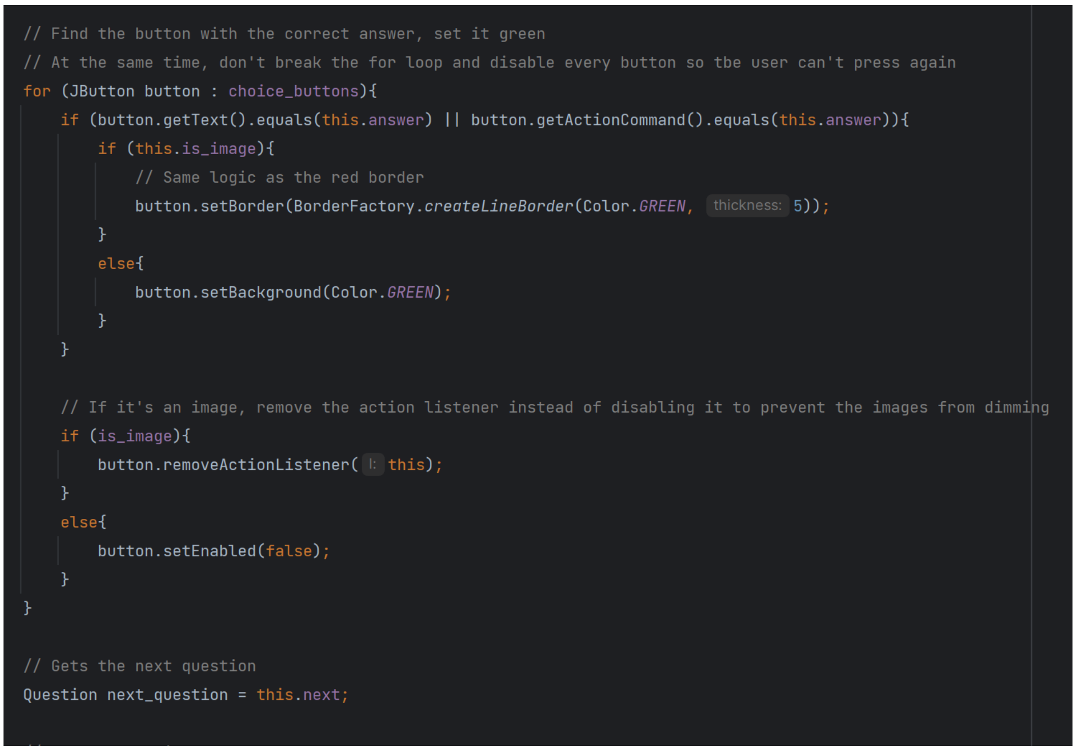
```java
@Override
public void actionPerformed(ActionEvent e) {
    // Get the source of the button press
    JButton button_pressed = (JButton) e.getSource();

    // If the button is the one to play audio
    if (button_pressed.getActionCommand().equals("play-audio")){
        try {
            // If the audio is already open and is running, restart it and stop it for a second
            if (clip != null && clip.isOpen()) {
                clip.setFramePosition(0);
                clip.stop();
                Thread.sleep( millis: 1000);
            } else {
                // If the audio isn't open yet, get the clip and open it
                clip = AudioSystem.getClip();
                clip.open(this.audio);
            }

            // Play the audio
            clip.start();
        } catch (LineUnavailableException | IOException | InterruptedException ex) {
            throw new RuntimeException(ex);
        }
    }
```

Here is the first part. I store the JButton that was clicked into button_pressed, and check if it is a button for playing audio. If it is, then the code showed above will be run. If it's the first time the button is pressed, I get the clip, store it into the clip attribute, and open the audio. Afterwards, the audio is played. Otherwise, I restart the audio, stop it, and give a 1 second delay before the audio plays again. I tried it without the delay, but it felt a lot better with the delay so that you can get your bearings. Now, if the button is not to play audio, and is one of the choice ones instead.



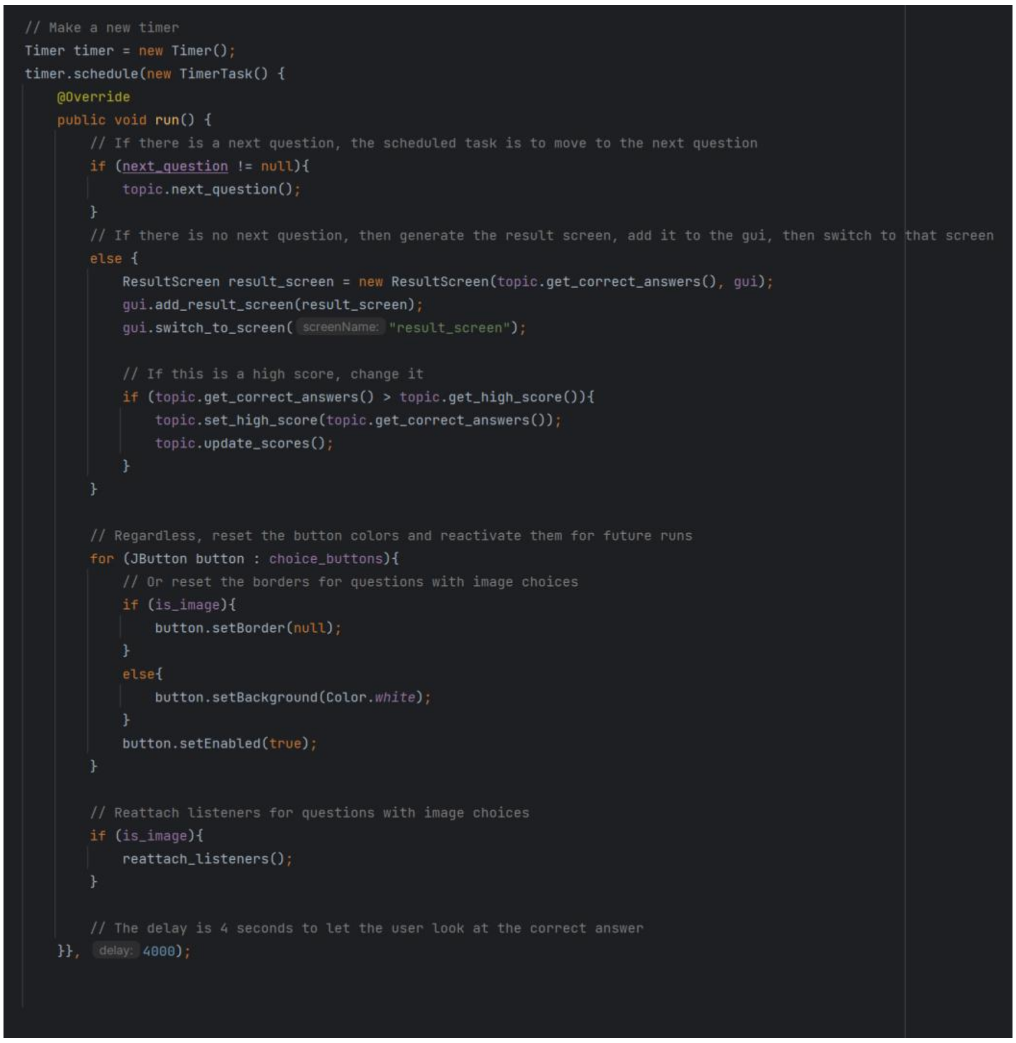
```java
// Else, the button is when the user chooses one of the choice buttons
else {
    // Stops the audio from playing in an audio question
    if (clip != null){
        clip.stop();
    }

    // If the button that's pressed is wrong, change its background to red
    if (!button_pressed.getText().equals(this.answer) && !button_pressed.getActionCommand().equals(this.answer)){
        if (this.is_image){
            // If it's an image, setting the background to red won't work, so give it a red border instead
            button_pressed.setBorder(BorderFactory.createLineBorder(Color.RED,  thickness: 5));
        }
        else{
            button_pressed.setBackground(Color.RED);
        }
    }

    // Else, the answer is correct and the number of correct answers is incremented
    else {
        topic.increment();
    }
```

If there is an audio playing, the audio is stopped. Next, it checks if the answer is correct or not. I only need to check if it is wrong though, because here is the thing. If you answer incorrectly, the button will turn red and the correct button will turn green. If you answer correctly, the button will turn green and that’s it. Regardless of whether or not the user answers correctly, the correct button will always turn green. I just need to check and see if it’s wrong, then turn it red. For buttons that contain text, I get its text and see if it is the same as the answer. For the buttons that are images, I get its action command (which is the image path) and check if it is the same as an answer. I use the AND logical operator here because depending on which type of choice the question is, one of the two conditions will always be true, and only the relevant one needs to be checked. Then, I change the button red for text choices and apply a red border for image choices. If the answer is correct, I call the increment method from the Topic class. This simply increments the number of correct answers for this run and the total number of correct answers.



```
// Find the button with the correct answer, set it green
// At the same time, don't break the for loop and disable every button so tbe user can't press again
for (JButton button : choice_buttons){
    if (button.getText().equals(this.answer) || button.getActionCommand().equals(this.answer)){
        if (this.is_image){
            // Same logic as the red border
            button.setBorder(BorderFactory.createLineBorder(Color.GREEN,  thickness: 5));
        }
        else{
            button.setBackground(Color.GREEN);
        }
    }

    // If it's an image, remove the action listener instead of disabling it to prevent the images from dimming
    if (is_image){
        button.removeActionListener( l: this);
    }
    else{
        button.setEnabled(false);
    }
}

// Gets the next question
Question next_question = this.next;
```

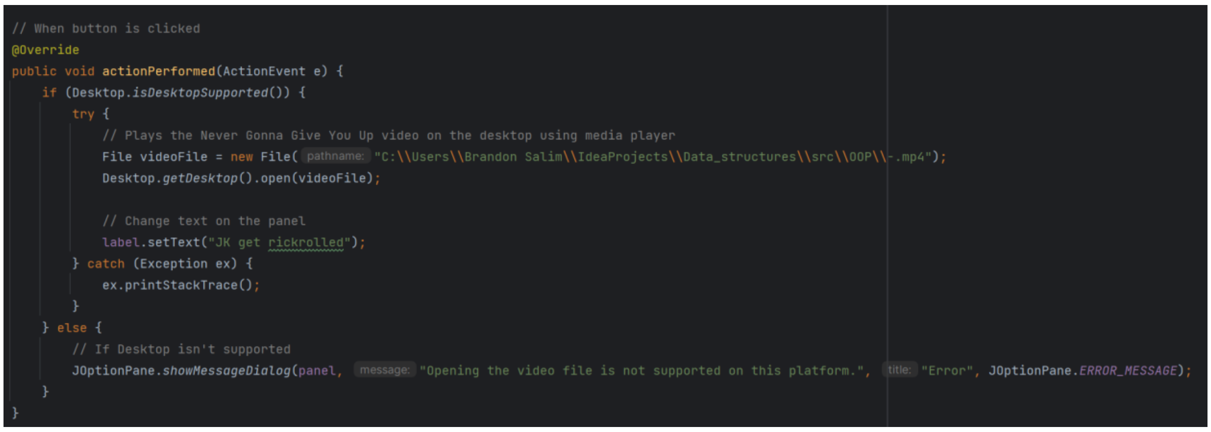
This next section is why I needed to add each button to the choice_buttons list back in the method that creates the choice panel. I loop through each button, and checks to see if it’s the correct answer. I use the OR operator here because one of the two conditions will always be false depending on the choice type. If the choices are images, the correct button will get a green border, and if it’s not, the button will turn green. Afterwards, I need to disable the buttons so that users don’t just spam them after answering. This part runs for every button. For normal text buttons, I simply disable the buttons, which is a lot simpler. However, when I disable buttons, the colours dim. This works perfectly fine for choices with text, and I say it even looks nice, but dimmed images means it’s hard to see the other choices if the user wants to see what the correct answer is. So instead of disabling the buttons, I remove the action listener momentarily. Lastly, I store the next attribute into the next_question variable. Here’s the next (and last) part:



```
// Make a new timer
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        // If there is a next question, the scheduled task is to move to the next question
        if (next_question != null){
            topic.next_question();
        }
        // If there is no next question, then generate the result screen, add it to the gui, then switch to that screen
        else {
            ResultScreen result_screen = new ResultScreen(topic.get_correct_answers(), gui);
            gui.add_result_screen(result_screen);
            gui.switch_to_screen( screenName: "result_screen");

            // If this is a high score, change it
            if (topic.get_correct_answers() > topic.get_high_score()){
                topic.set_high_score(topic.get_correct_answers());
                topic.update_scores();
            }
        }

        // Regardless, reset the button colors and reactivate them for future runs
        for (JButton button : choice_buttons){
            // Or reset the borders for questions with image choices
            if (is_image){
                button.setBorder(null);
            }
            else{
                button.setBackground(Color.white);
            }
            button.setEnabled(true);
        }

        // Reattach listeners for questions with image choices
        if (is_image){
            reattach_listeners();
        }

        // The delay is 4 seconds to let the user look at the correct answer
    }}, delay: 4000);
```

I create a new timer and timer task, then override the run method. If a next question exists, then this current question isn't the last tone, and the next_question method is called from the Topics class. If the next question is null, then this question is the last, and it makes the result screen. To create a result screen, it needs the number of correct answers, which is obtained from the getter method in the Topic class. Afterwards, I add the result screen to the main card layout panel, and switch to the result screen. Additionally, it checks to see whether or not this run is a high score. If it is, it calls the method from the Topic class to set the high score, and the update_scores method from the Topic class. The first is just a setter for the high_score attribute in the Topic class, while the second one calls the method from the TopicScreen class that updates that panel and the text file. The next part is to reset everything. For each button in the choice_buttons list, the buttons are returned to their original form. For image choices, the border is removed, and for the text choices, the buttons are set back to white. Additionally, the buttons are enabled again. Lastly, to make the image buttons work again, I call the reattach_listeners method. The name explains itself. It just loops through each button and reattach the action listeners. I decided that the delay is 4s. It isn't too long, and gives an adequate amount of time to look at correct answer.

The ResultScreen class and the Screen abstract class are quite ordinary, so the last thing I'll show it this, from the Rickroll class.



```java
// When button is clicked
@Override
public void actionPerformed(ActionEvent e) {
    if (Desktop.isDesktopSupported()) {
        try {
            // Plays the Never Gonna Give You Up video on the desktop using media player
            File videoFile = new File( pathname: "C:\\Users\\Brandon Salim\\IdeaProjects\\Data_structures\\src\\OOP\\-.mp4");
            Desktop.getDesktop().open(videoFile);

            // Change text on the panel
            label.setText("JK get rickrolled");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } else {
        // If Desktop isn't supported
        JOptionPane.showMessageDialog(panel,  message: "Opening the video file is not supported on this platform.",  title: "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

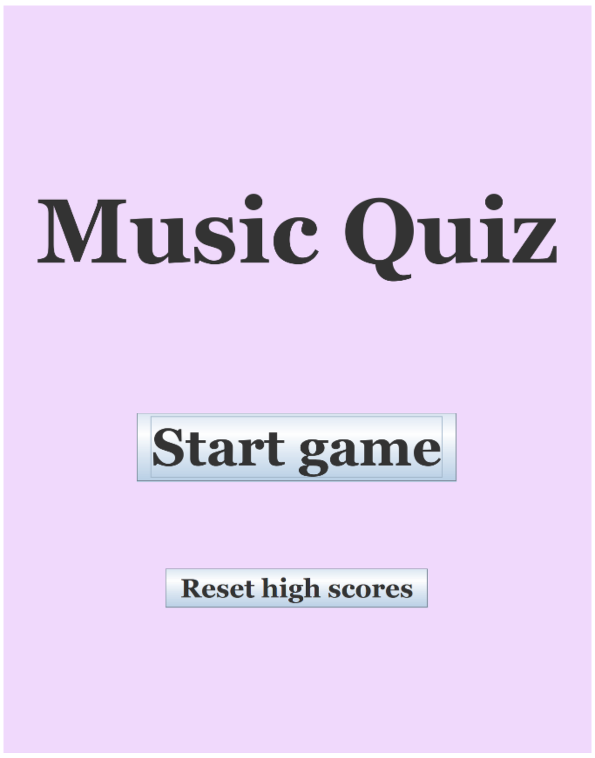
I couldn’t find a way to make the video pop up on the gui with Java Swing, but I ended up finding a better alternative where the media player just pops up and takes up the whole screen instead of just being on the gui. So, here is the actionPerformed method on the rickroll class. It makes a new File object from the class, then opens the video file using the desktop. The text on the gui is changed, and the video is opened with the default application to open mp4s.

That’s it for the explanation for the code. Hopefully, it’s clear enough.

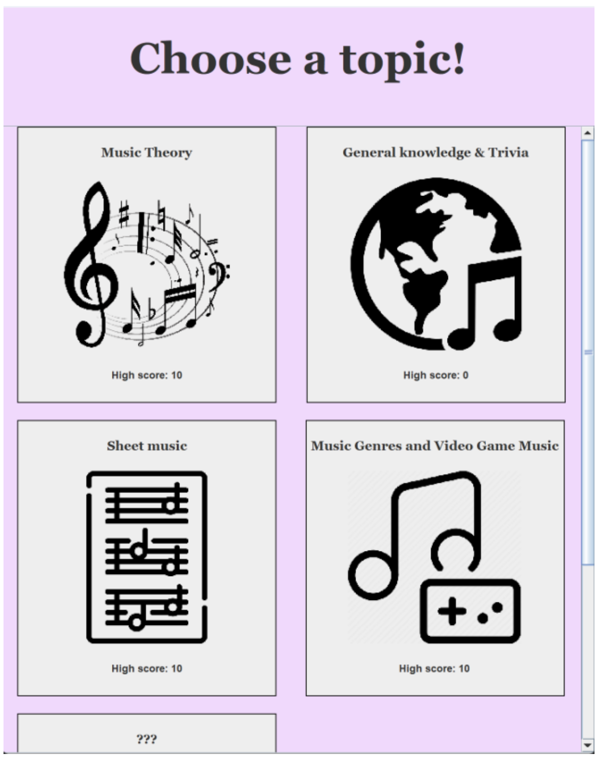
IV. Evidence

In this section, I will show a screenshot for each part of the program, and include a link for a video demonstration in the end.

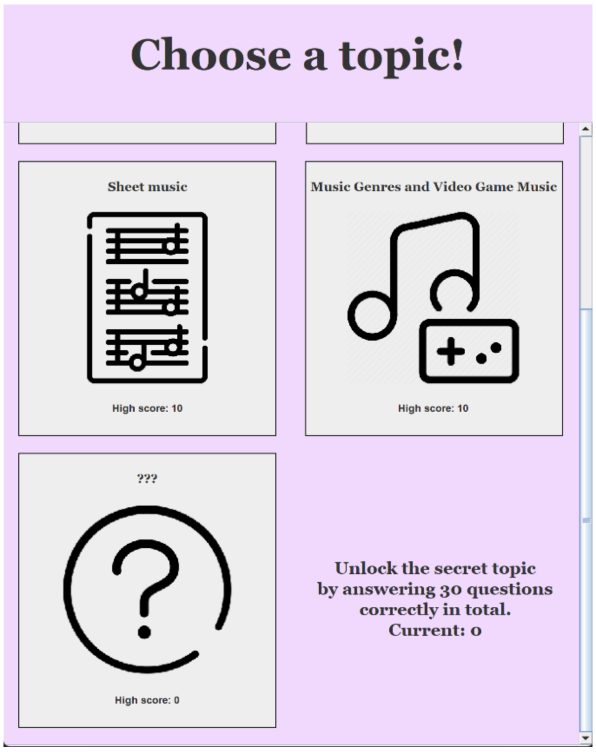
Title screen



Topic Screen (1)



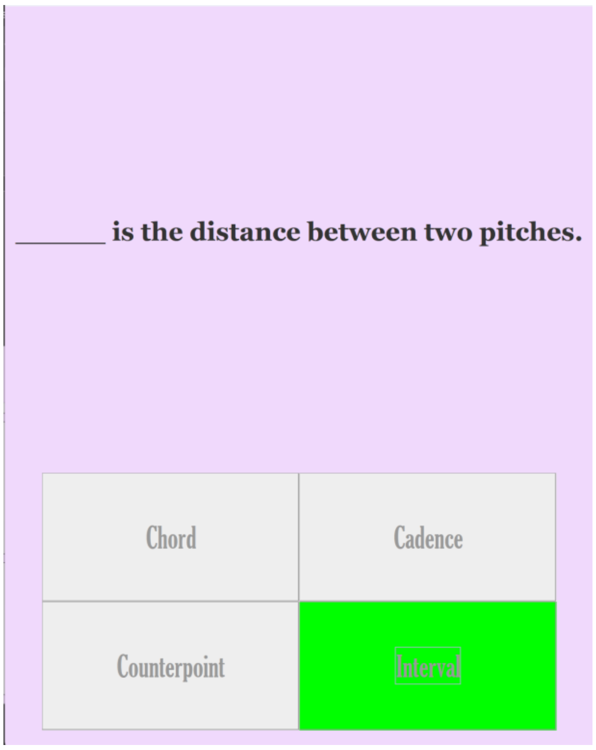
Topic Screen (2)



Choose a topic!

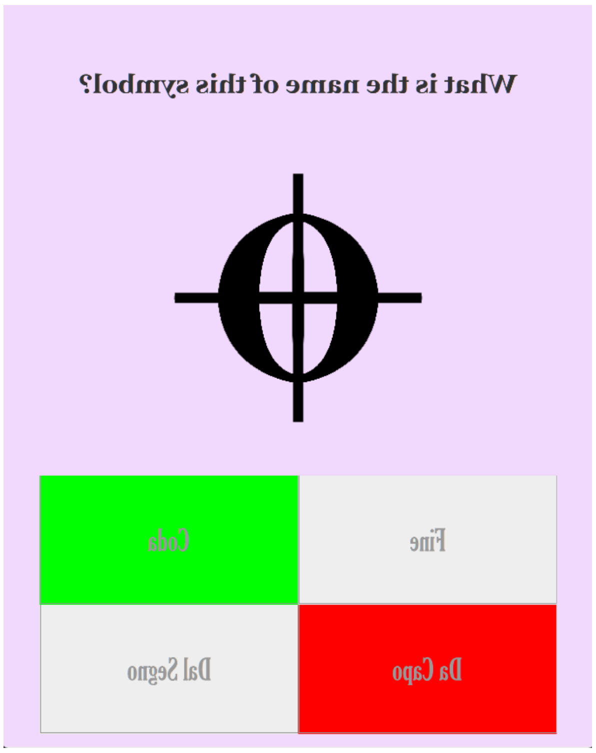
Unlock the secret topic by answering 30 questions correctly in total. Current: 0

Question (text, 4 choices, correct answer)



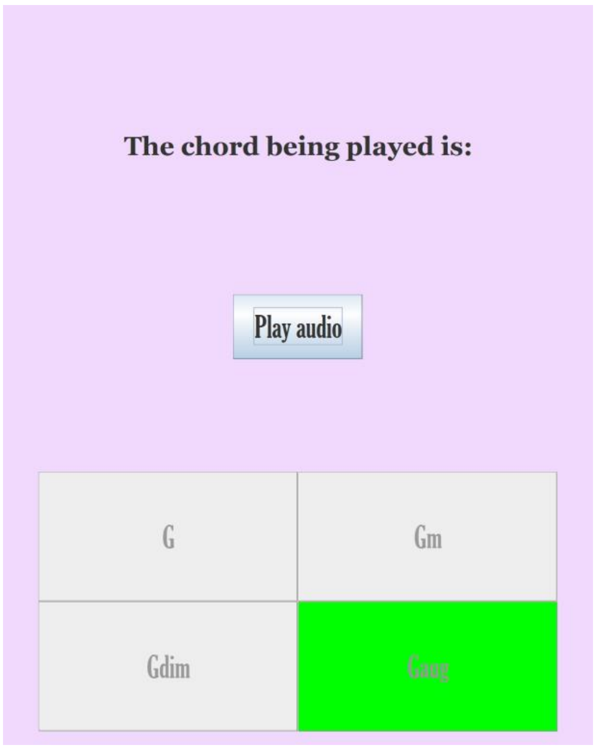
_______ is the distance between two pitches.

Question (image, 4 choices, wrong answer)



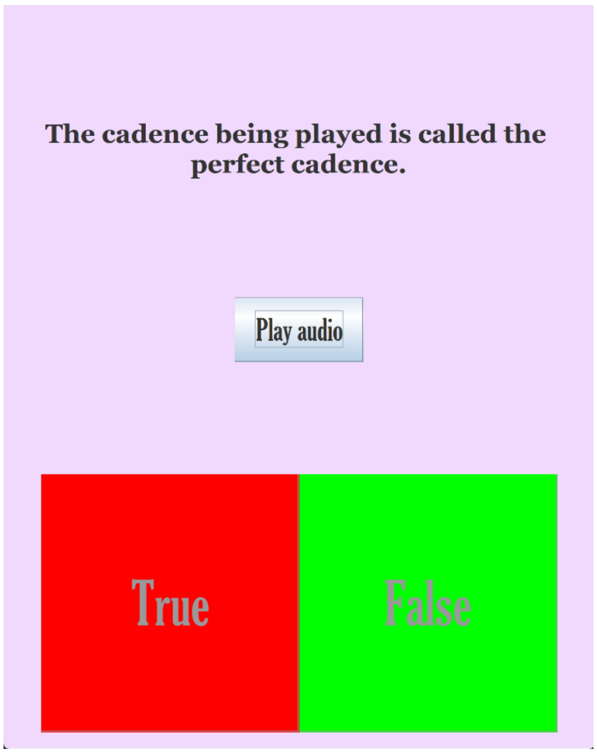
What is the name of this symbol?

Question (audio, 4 choices, correct answer)

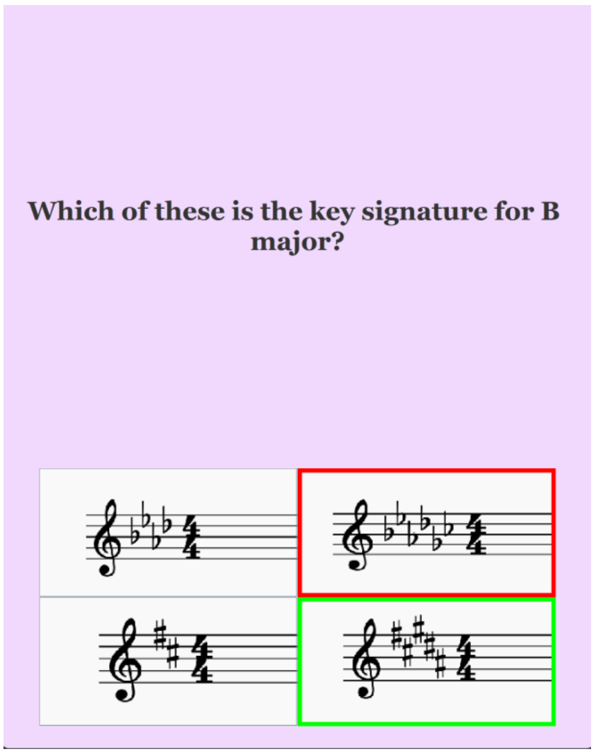


The chord being played is:

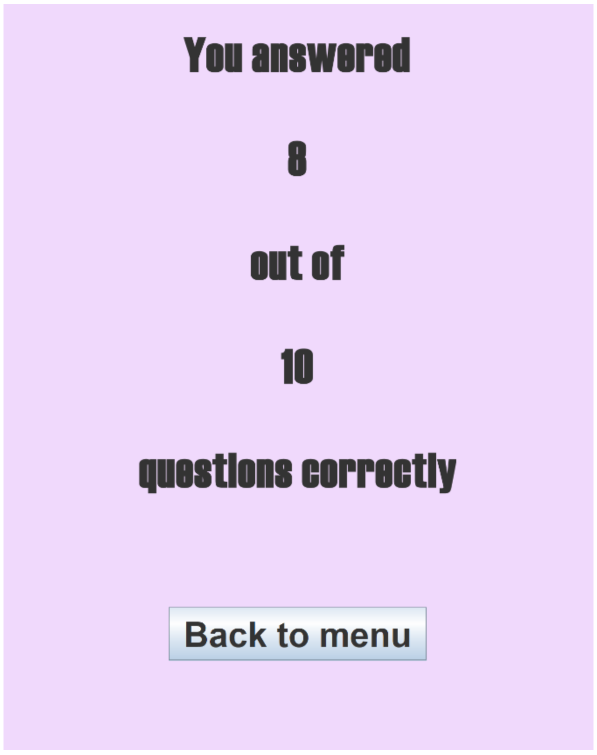
Question (audio, 2 choices, wrong answer)


The cadence being played is called the perfect cadence.

Question (text, image choices, wrong answer)


Which of these is the key signature for B major?

Result screen (8 correct out of 10)


You answered

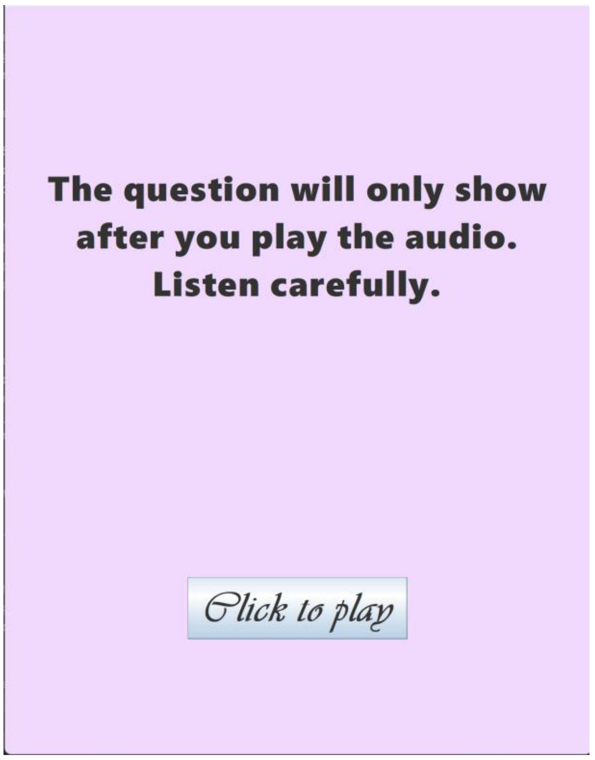
8

out of

10

questions correctly

Screen when you click secret topic


The question will only show after you play the audio. Listen carefully.

Video link:
https://drive.google.com/file/d/1kU-E8cpnfT6hzBANE_hK9Uq284sp6wbx/view?usp=sharing

Github link: https://github.com/BrandonSalimTheHuman/OOP-Final-Project

V. Personal Thoughts

Honestly, I was about sceptical about this project. I took quite a long time to even determine what to make this semester, since I didn't really have any ideas. After hours and hours of trying to find something to make for the project, and even trying to ask chatgpt to generate one hundred topics, nothing really seemed plausible. As I mentioned at the start, I thought of this when I was looking at a quizziz, but for quite a while, I wasn't sure whether or not this could be a valid project. On the surface, it seemed quite simple. Especially so when I heard about what the other people in the class are making. For instance, I remember Michael said he's going with the Pokemon idea he had a few weeks prior for the project. The idea is hilarious, and it's great to see him go ahead with it. But in all honesty, I just felt more and more doubtful about my idea, afraid that it's just too simple. In the end, I asked a couple other people what they think about my project idea. Unlike me, they said that it'll be fine, and that it's definitely a valid project idea. At that point, I knew that I only have around 2 weeks to finish the project, so I just went on with it, and that's how this project came to be.

And honestly, they were right. As I started doing it, I realized that this seemingly simple project has a lot more going for it than I initially thought, and I ran into a lot more issues that imagined before starting. I wasn't sure how to switch screens, and when I saw how the card layout works, I had to think about how I can organize the screens. The title screen was easy, but I was immediately stumped at the topic screen because I couldn't seem to just place things on the screen where I wanted to be. I ended up learning about different layouts, like the grid bag layout to centre stuff, the grid layout that can have a dynamic number of rows, the box layout, etc. It took a surprising amount of time to actual think of a valid way to use the layouts to form the topic screen, albeit it looking very simple in my head. After that, the biggest challenge was the Question class. At first, I settled on the three constructors and made them. I already learnt how to get images from the topic screen, so the layout itself wasn't too bad this time. However, it took a while to get the audio working. It didn't want to start at first, then it wouldn't stop, then it skips first note while repeating, etc. Afterwards, I had to think of a way to show the questions. This time I knew that I can use the card layout, but I realized I need to put a card layout into a card layout. A few hours of tinkering later, and I ended up making it work by putting the gui instance in all of the classes that needs to switch screens, and by making the topic screen switch questions. The result screen class wasn't that bad to make, but making sure that only the last question generates it did take a while. I also learnt about the timer and timer task here. This is the point where I also started to realize how absurd of an idea it was for me to make 120 questions on music myself. Too late to turn back though, so over 3 days, I made 30 questions for each topic. I thought of basic music theory from what I know, took some inspiration for the questions from songs I enjoy, and even some funnier questions that references the class in some way. For the images and some of the audio, I ended up opening Musescore after a while, to create the sheet music. Afterwards, I thought I was almost done, just missing one more topic. And since the topic ended up being sheet music, I had to reconsider the Question class again. I added the Boolean and added several methods for the image to work, as well as adding lines is all other methods to account for the image questions. However, it did end up being a good thing, because while I was reevaluating the code in that class, I managed to spot similarities and reduce the number of lines by making methods instead. I also had to open Musescore again and export the sheet music as a png nearly 80 times, not including the audio. Definitely quite painful, but it's just time-consuming more than anything. Other than that, I honestly don't even remember when I thought of the rickroll. I'm pretty sure I just thought of it while I was making the questions, and just stopped making them and dedicated the next couple hours to make the rickroll work.

Anyways, that’s it for this report. I definitely learnt a lot about Java Swing in this project, and how layouts can get really complicated, since you can’t just drag something and place it on the screen wherever you want. This project also made me more comfortable using Java, and I do feel that I understand the logic behind Java more. But other than that, I also learnt that I should try not be as paranoid. I took so long to go with this project because I was scared that it won’t work out, because it’ll be too short and whatnot. In the end however, the project was quite challenging, and took a lot more effort than I expected. Hopefully I can learn to just go with my guts more next time, and to just go with an idea if I want to do it.