Stage 1 – Creating sequencer:

In this stage I will be creating the main sequencer GUI and the individual instrument GUI. I will also add various features such as playback, adding notes, and a data structure for the sequencer. By the end of this stage, the user should be able to click notes into an instruments sequence and then play this back. I am doing this first as it is crucial for the rest of the program to operate properly, and without this being implemented, I will not be able to perform iterative tests on the rest of the program.

Design:

Algorithm design:

- For the GUI structures, there is not any pseudocode or flow-charts to add, since I can implement different features of the GUI straight into it through IntelliJ UI forms.
- The data structure for the timeline will be a 2D array, so that I can sort notes by their place on the timeline (first dimension) and pitch (second dimension). This will make playback easier as this data will simply be read and converted into a sequence:

```
public class Instrument{
        private timeLine = new int[128][1024];
        private int type;
        Private boolean playing = false;
        Private boolean paused = false;

        public Instrument(int typeNum){
                                this.timeLine = new int[128][1024];
                this.type = typeNum;
        }

        public void addNote(int pitch, int place){
                timeLine[place][pitch]  = 1;
        }

        public void removeNote(int pitch, int place){
                timeLine[place][pitch] = 0;
        }

        public void play(int currentBeat){
        If(!playing && !paused){
                        Synth.open()
                        Synth.play(timeLine, type, currentBeat)
                }
                //the actual code is a lot more complex than this, so I will learn how to
        do specific functions for this method during development.
        }
        Public Array getTimeLine(){
            Return this.timeLine;
        }
        Public int getType(){
            Return this.type;
        }
        Public int getNoteState(int pitch, int place){
            Return this.timeLine[place][pitch];
        }
        Public boolean isPlaying(){
        Return playing;
        }
        Public boolean isPaused(){
        Return paused;
        }
        Public void stopAndRewind(){
        Playing = false;
        Paused = false;
        }
        Public void pause(){
        Paused = true;
        }
        Public void resume(){
        Paused = false;
        }
}
```

- On the main class, I will create listeners, so that when the user clicks on the grid to add/remove a note, the addNote/removeNote method will be run, and the block on the grid will fill to let the user know that it has been added:

```
Public class gridBox{

    Private int place;

    Private int pitch;


    Public gridBox(int pitch, int place){

        This.place =  place;

        This.pitch =  pitch;

    }

    Public void actionPerformed(ActionEvent e){
        If(Instrument.getNoteState(this.place, this.pitch){
            This.gridBox.setFill("Black");
            instrument.addNote(this.place, this.pitch);
        }
        Else{
            This.gridBox.setFill("White");
            instrument.removeNote(this.place, this.pitch);
        }

    }

}
```

Data:

- Below is the data dictionary that I will use for this stage, it includes all methods and attributes that will be used during the programming of the sequencer. I created this so that I can reference this in my future stages to easier gather information about attributes, methods, and what they do.

Attributes:

Name	Type	Description	Held in class:
timeLine[128][1024]	Array of integers	Holds the current timeline state for each instrument. A “1” refers to a note of a pitch and place on the timeline, and “0” is a lack of a note.	Instrument
type	Integer	Holds a reference number as to the instrument that is being composed (i.e. 1 = piano, 2 = guitar, etc)	Instrument
place	Integer	Holds the place within the timeline of each gridBox, so that notes can be added easier.	gridBox
pitch	integer	Holds the pitch of each gridBox, so that notes can be added easier.	gridBox
Active	Boolean	True if the note exists in the grid cell	gridBox
playing	boolean	True if the sequencer is currently playing	Instrument
paused	boolean	True if the sequencer has been paused	instrument

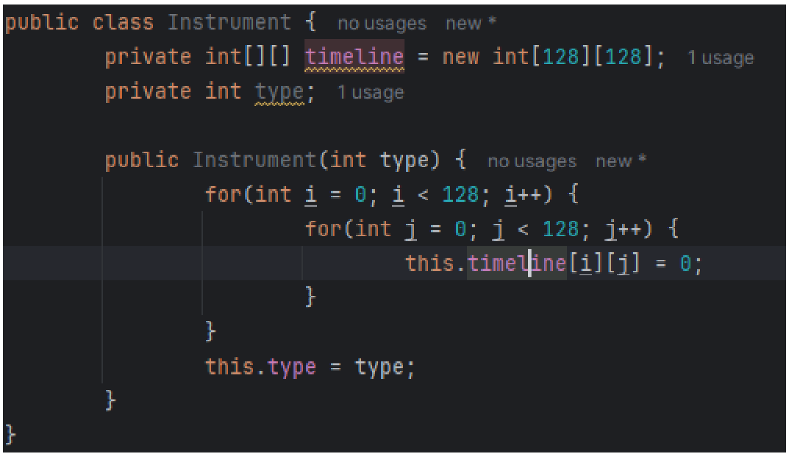
Methods:

Name	Return type	Description	Held in class:
addNote	Void	Used to add a note onto the timeline	Instrument
removeNote	Void	Used to remove a note from the timeline.	Instrument
Play	Void	Used to convert the timeline into an mp3 file, and then play it	Instrument
setType	Void	Setter for the type attribute	Instrumnent
getType	Integer	Getter for type attribute	Instrument

getNoteState	Integer	Getter for the state of a current note (1 or 0) in the timeLine	Instrument
actionPerformed	Void	Does necessary actions when a note has been added or removed from the sequence.	gridBox
Activate	Void	Sets a gridBox to active (note)	gridBox
Deactivate	Void	Sets a gridBox to deactive (no note)	gridBox
getActive	Boolean	Getter for active attribute	gridBox
getPlace	Int	Getter for place attribute	gridBox
getPitch	Int	Getter for pitch attribute	gridBox
isPlaying	boolean	Getter for playing attribute	Instrument
isPaused	boolean	Getter for paused attribute	Instrument
stopAndRewind	void	Does necessary actions for rewinding the sequence	Instrument
pause	void	Pauses the sequence	Instrument
resume	void	Resumes the sequence	instrument

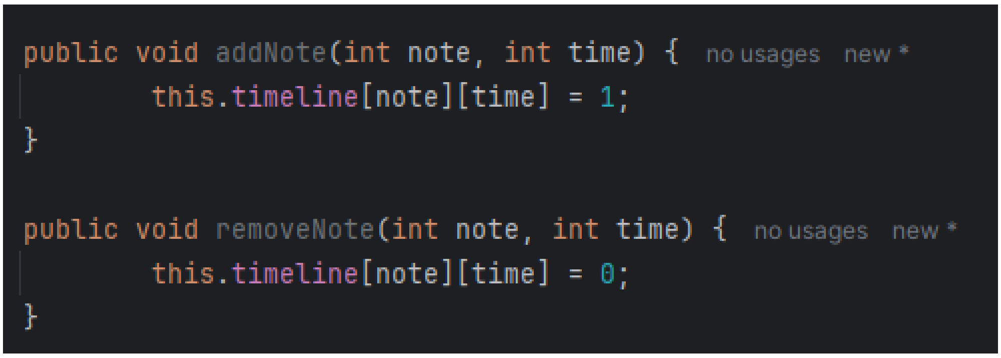
Development:

I first started by adding the GUI form and Instrument class. I did not add any features to the GUI form yet, since focusing on the Instrument class was my priority. I added some simple attributes for type of instrument (an integer corresponding to an instrument) and the timeline (a 2D array of integers to represent the pitch and placement of a note). I also created a constructor for the Instrument class which declares all values within the timeline as 0, since there are no initial notes within the sequence. This constructor also declares the type of instrument as a value passed into the special method. I used a for loop to declare the timeline, since it allowed me to change all values within the array:



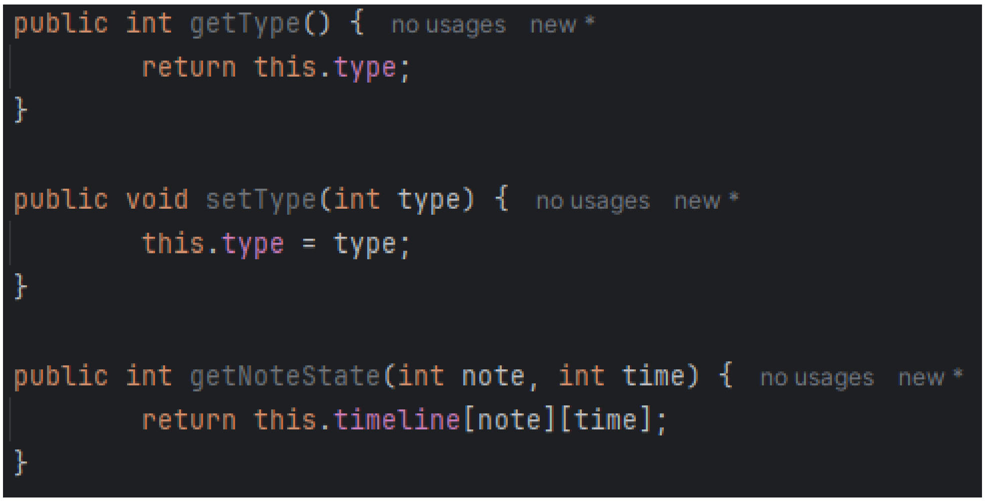
```
public class Instrument {  no usages  new *
        private int[][] timeline = new int[128][128];  1 usage
        private int type;  1 usage

        public Instrument(int type) {  no usages  new *
                for(int i = 0; i < 128; i++) {
                        for(int j = 0; j < 128; j++) {
                                this.timeline[i][j] = 0;
                        }
                }
                this.type = type;
        }
}
```

After this I created 2 more methods for adding/removing notes to the sequence. I made it so that each method would set a specific place and pitch in the timeLine attribute to 1. I did this so that we can reference these values in the array to create the actual sound file:



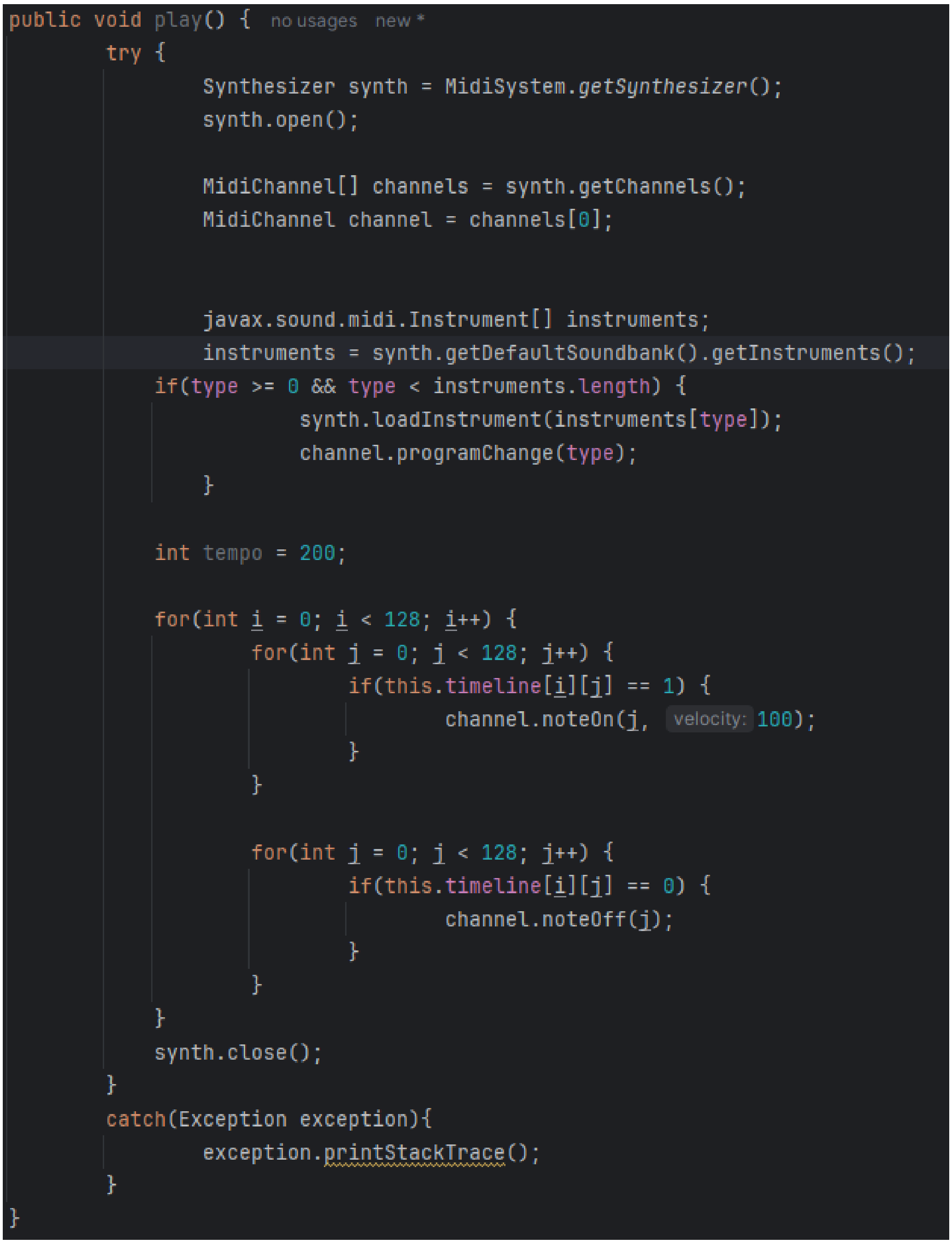
```
public void addNote(int note, int time) {  no usages  new *
        this.timeline[note][time] = 1;
}

public void removeNote(int note, int time) {  no usages  new *
        this.timeline[note][time] = 0;
}
```

I then made getters and setters for each of the attributes within the Instrument class. This will be useful during further programming for accessing private attributes within this class, or changing the value of certain attributes:



```
public int getType() {
        return this.type;
}

public void setType(int type) {
        this.type = type;
}

public int getNoteState(int note, int time) {
        return this.timeline[note][time];
}
```

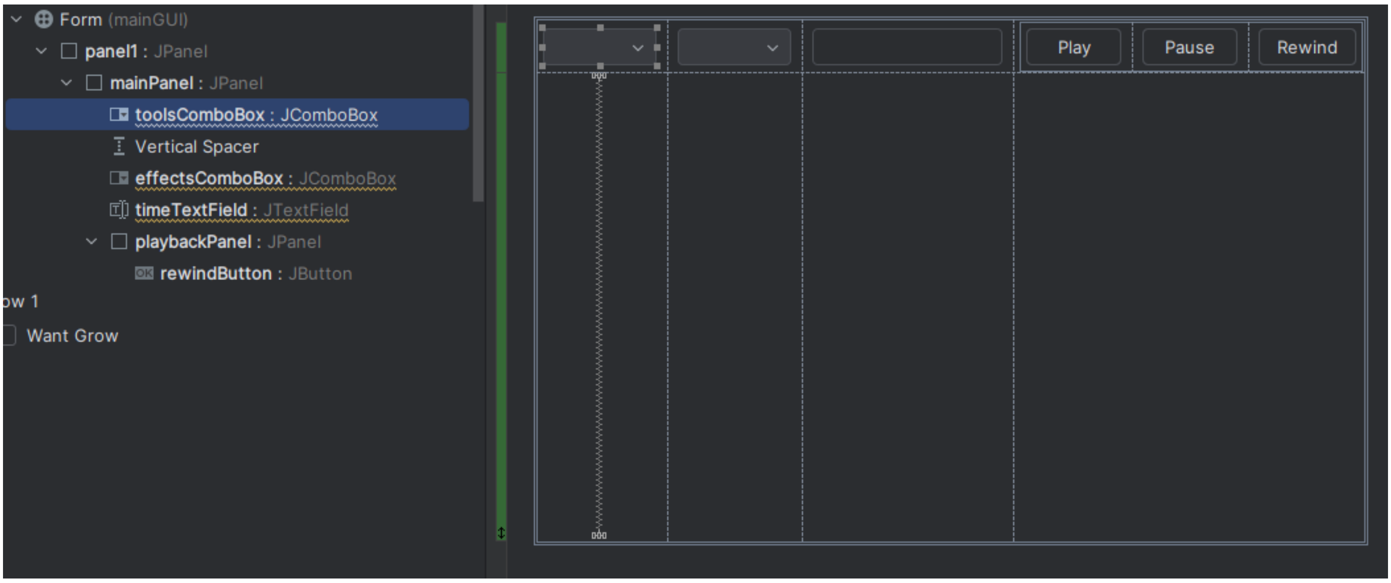
After that, I started creating the play method. For this I had to learn how to create and manipulate sound files, so I looked at REINTECH (https://reintech.io/blog/java-midi-programming-creating-manipulating-midi-data), and rememberjava (https://rememberjava.com/midi/2017/01/13/midi_basics.html) which taught me important techniques about using java’s MIDI synthesizer. This will allow me to use java’s MIDI synthesizer to create a playback of a sound file, based upon the state of each item in timeLine:



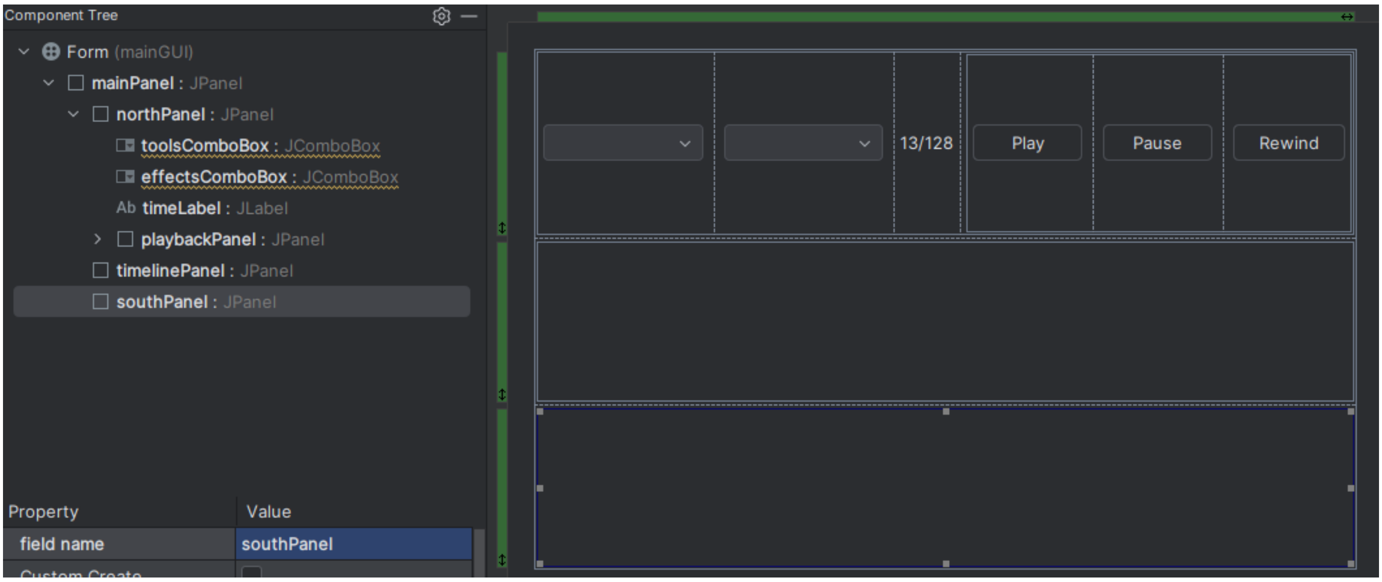
```
public void play() {  no usages  new *
        try {
                Synthesizer synth = MidiSystem.getSynthesizer();
                synth.open();

                MidiChannel[] channels = synth.getChannels();
                MidiChannel channel = channels[0];


                javax.sound.midi.Instrument[] instruments;
                instruments = synth.getDefaultSoundbank().getInstruments();
            if(type >= 0 && type < instruments.length) {
                        synth.loadInstrument(instruments[type]);
                        channel.programChange(type);
                }

            int tempo = 200;

            for(int i = 0; i < 128; i++) {
                    for(int j = 0; j < 128; j++) {
                            if(this.timeline[i][j] == 1) {
                                    channel.noteOn(j,  velocity: 100);
                            }
                    }

                    for(int j = 0; j < 128; j++) {
                            if(this.timeline[i][j] == 0) {
                                    channel.noteOff(j);
                            }
                    }
            }
            synth.close();
        }
        catch(Exception exception){
                exception.printStackTrace();
        }
}
```

This is only a draft of the final code and may change based on other parts that I code (E.g. changing tempo or multi-sequencing). This method is wrapped in a try block so any errors can be caught. The first thing the play method does is creating a MIDI synthesizer and opening it and then uses the first channel to use for playback of the instrument. The instruments sound is then set based on the type attribute, to a sound within the synthesizer's sound bank. The timeline is then played through by turning noteOn for every 1 within the timeLine attribute, and noteOff after the note has been played.

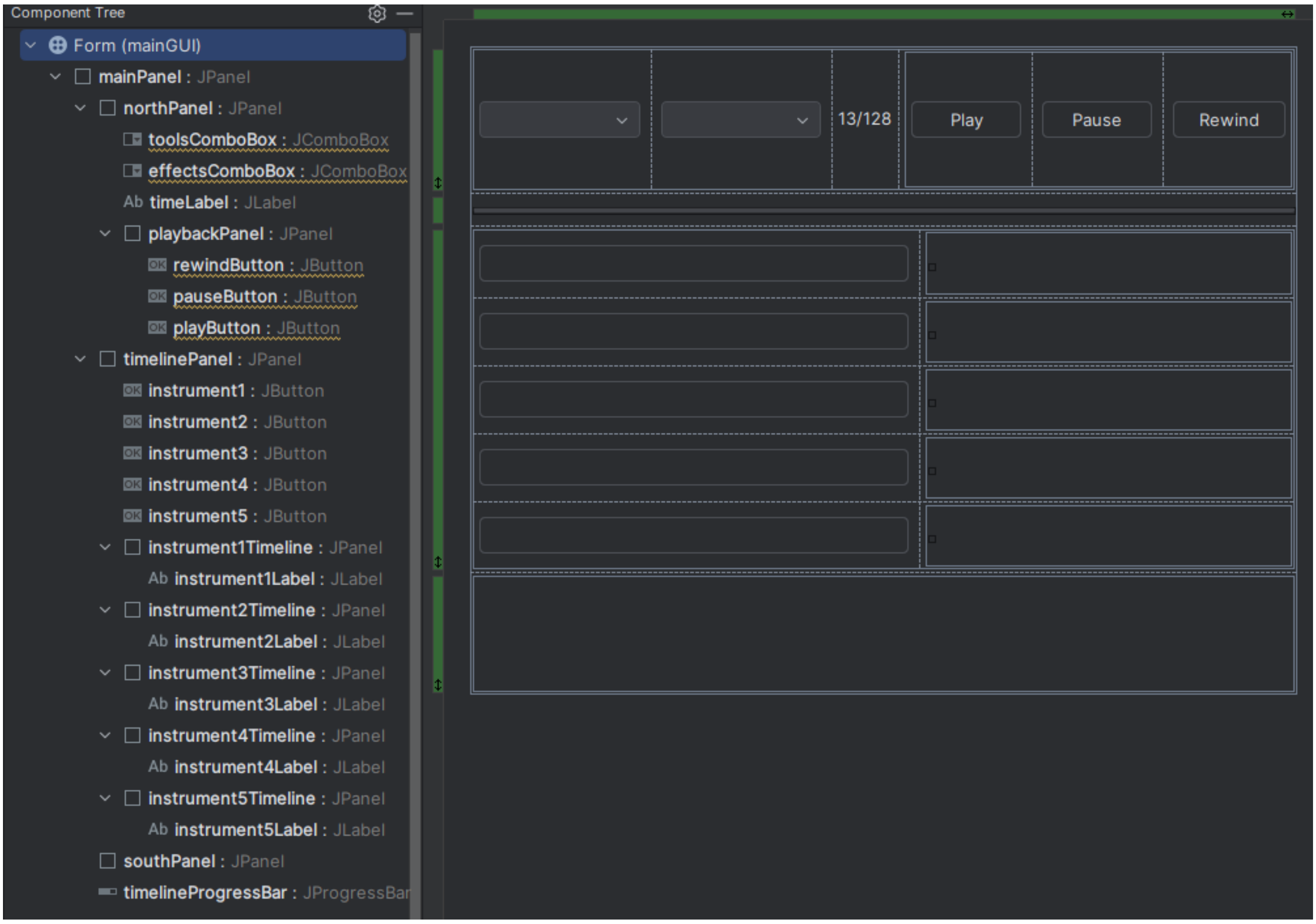
The next thing that I constructed was the main sequencer GUI. I did this my adding several JButtons, JComboBox's, and JTextField's onto a JPanel on my mainGUI. I gave these relevant names and places on the GUI with respect to my initial GUI designs. I did this so that there is a main navigation GUI for the user to access. Here is the current state of the GUI:



I then did some rearrangements to this mainGUI by adding a northPanel, southPanel, and timelinePanel (all JPanels). I did this because it would give my GUI a better structure for me to add other components. This is how it looked after I did so:

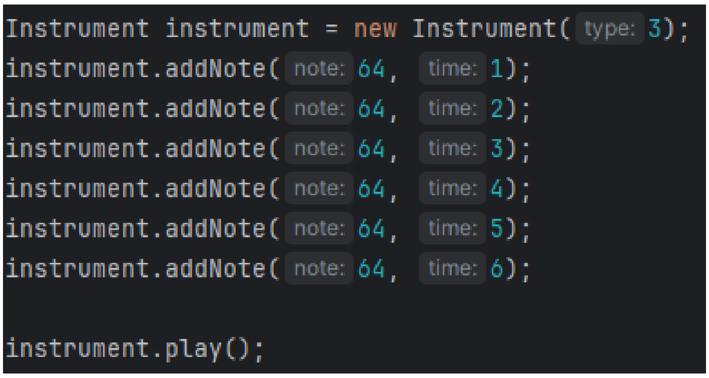


After that I added more features for the timeline, such as a button at the start of each line which the user could click to take them to that instrument's individual timeline, and JLabels for displaying each sequence. In addition to that I added a timelineProgressBar to display how far into the song has been played. I added all these features by following my GUI design that I created prior to this development and did this so that the user has a good interface for the user to access all different parts of the GUI and access to all features. This is how it looked after this development:



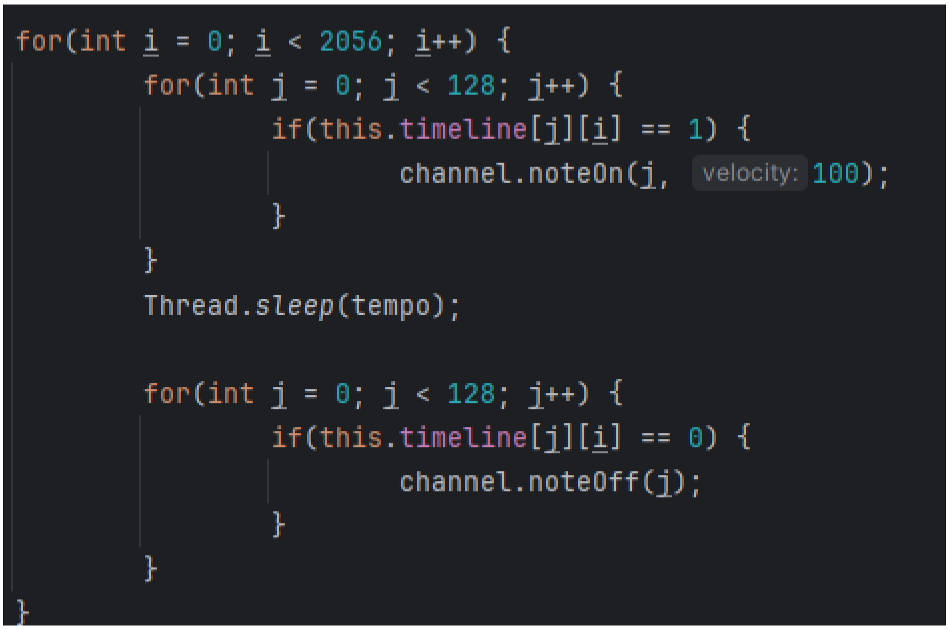
```
Component Tree
Form (mainGUI)
  mainPanel : JPanel
    northPanel : JPanel
      toolsComboBox : JComboBox
      effectsComboBox : JComboBox
      timeLabel : JLabel
      playbackPanel : JPanel
        rewindButton : JButton
        pauseButton : JButton
        playButton : JButton
    timelinePanel : JPanel
      instrument1 : JButton
      instrument2 : JButton
      instrument3 : JButton
      instrument4 : JButton
      instrument5 : JButton
      instrument1Timeline : JPanel
        instrument1Label : JLabel
      instrument2Timeline : JPanel
        instrument2Label : JLabel
      instrument3Timeline : JPanel
        instrument3Label : JLabel
      instrument4Timeline : JPanel
        instrument4Label : JLabel
      instrument5Timeline : JPanel
        instrument5Label : JLabel
    southPanel : JPanel
    timelineProgressBar : JProgressBar
```

Before I started anything else, I tested my play method to make sure that the correct notes were playing in the correct order. I did this because it is fundamental to the rest of the program working. This is the code I ran:

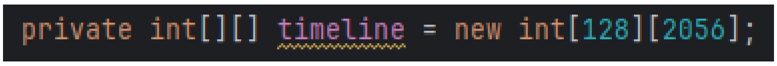


```
Instrument instrument = new Instrument( type: 3);
instrument.addNote( note: 64,  time: 1);
instrument.addNote( note: 64,  time: 2);
instrument.addNote( note: 64,  time: 3);
instrument.addNote( note: 64,  time: 4);
instrument.addNote( note: 64,  time: 5);
instrument.addNote( note: 64,  time: 6);

instrument.play();
```

I noticed that all notes were being played at the same time but at different pitches. This made me check that I had created the for loops correctly when initially coding the play method, and I had not. In addition to this, I changed the number of items in the timeLine array to add more space for longer sequences to be made. This will hopefully increase usability in the long run. Here is what the new code that I changed/added looks like:

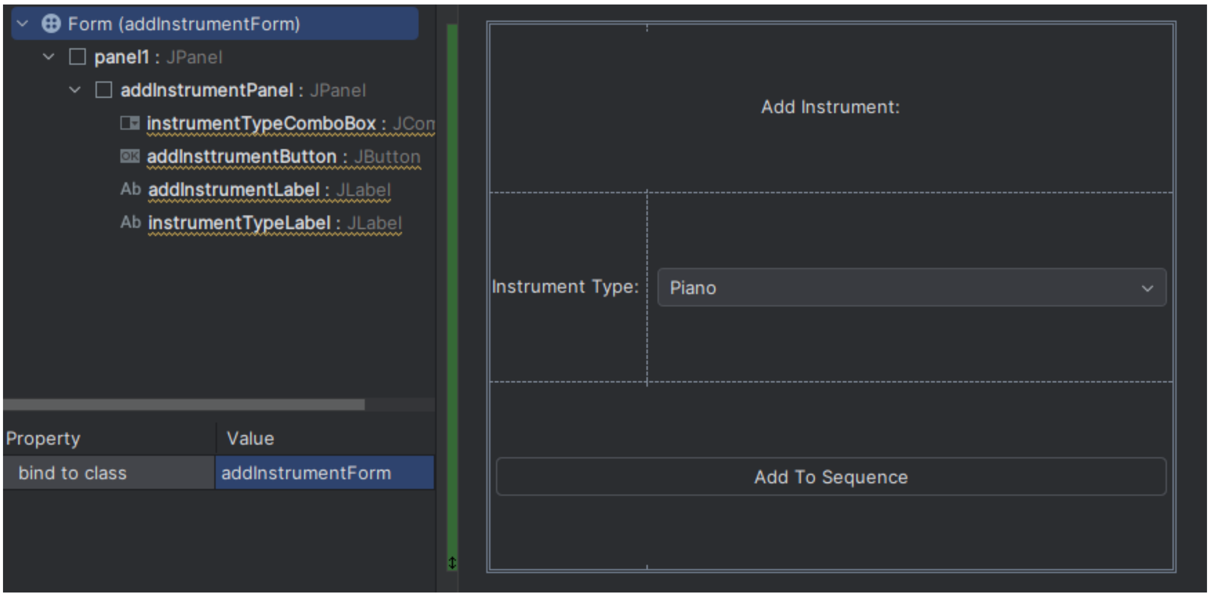


```
for(int i = 0; i < 2056; i++) {
        for(int j = 0; j < 128; j++) {
                if(this.timeline[j][i] == 1) {
                        channel.noteOn(j, velocity: 100);
                }
        }
        Thread.sleep(tempo);

        for(int j = 0; j < 128; j++) {
                if(this.timeline[j][i] == 0) {
                        channel.noteOff(j);
                }
        }
}
```



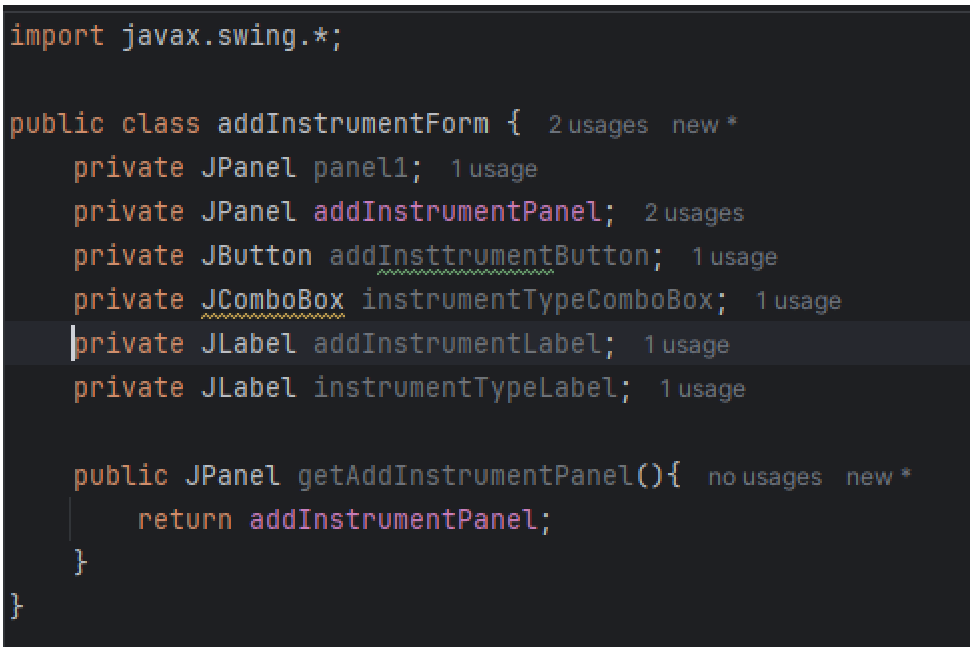
```
private int[][] timeline = new int[128][2056];
```

After that, I wanted to make a small UI for adding an instrument. The reason that this was not in my GUI designs is because it is quite simple and only contains a few components (JPanel, JButtons, JComboBox). I also made a getter for the main JPanel since it will allow me to access it from the main form. This is the first template and the code for the main JPanel's getter:



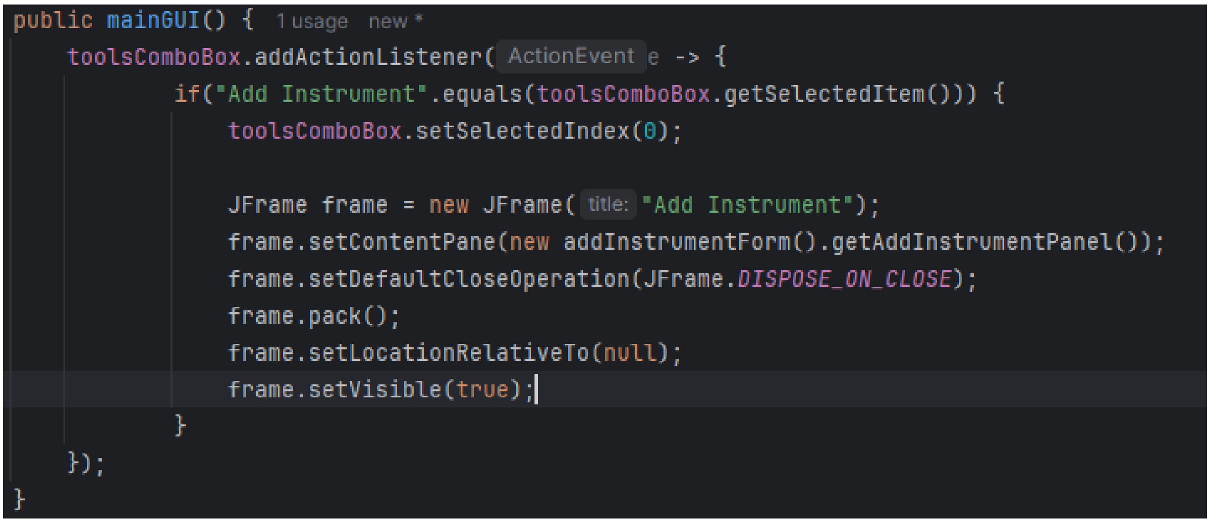
```
Form (addInstrumentForm)
    panel1 : JPanel
        addInstrumentPanel : JPanel
            instrumentTypeComboBox : JCon
            addInsttrumentButton : JButton
            addInstrumentLabel : JLabel
            instrumentTypeLabel : JLabel
```

Property	Value
bind to class	addInstrumentForm



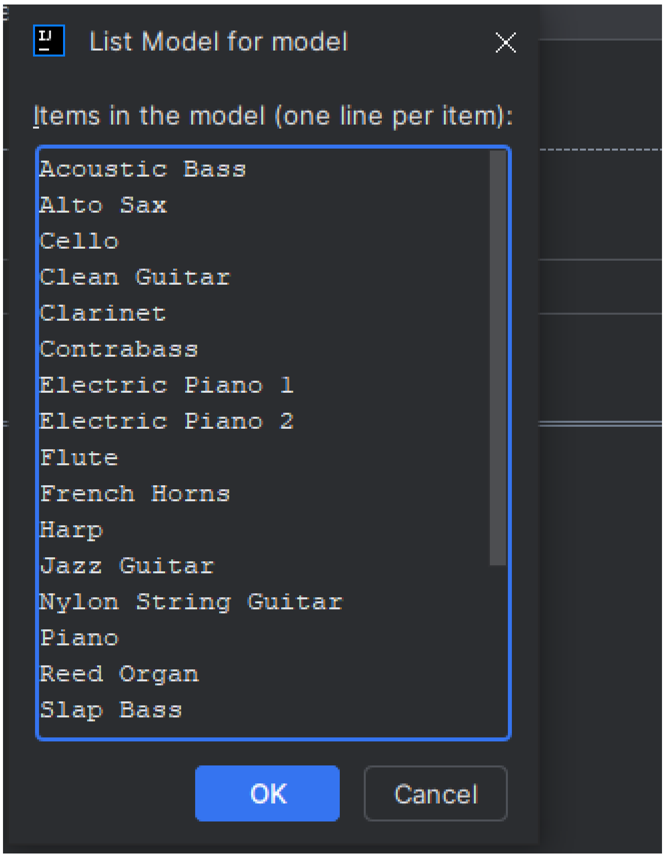
```
import javax.swing.*;

public class addInstrumentForm {  2 usages  new *
    private JPanel panel1;  1 usage
    private JPanel addInstrumentPanel;  2 usages
    private JButton addInsttrumentButton;  1 usage
    private JComboBox instrumentTypeComboBox;  1 usage
    private JLabel addInstrumentLabel;  1 usage
    private JLabel instrumentTypeLabel;  1 usage

    public JPanel getAddInstrumentPanel(){  no usages  new *
        return addInstrumentPanel;
    }
}
```

I then created an ActionListener for the toolsComboBox, which checks if the item state has changed to "Add Instrument". The code inside this sets the toolsComboBox back to Tools and opens the addInstrumentForm so that the user can add an instrument. To help me code this I referred to FAQ on the stackoverflow website (https://stackoverflow.com/questions/35821071/learning-guis-setcontentpane-method) which taught me how to set different forms to open. The reason I did this is to make manoeuvring around different forms easier for the user: This is what my new code that I added looks like:



```
public mainGUI() {  1 usage  new *
    toolsComboBox.addActionListener( ActionEvent e -> {
            if("Add Instrument".equals(toolsComboBox.getSelectedItem())) {
                toolsComboBox.setSelectedIndex(0);

                JFrame frame = new JFrame( title: "Add Instrument");
                frame.setContentPane(new addInstrumentForm().getAddInstrumentPanel());
                frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setVisible(true);
            }
    });
}
```

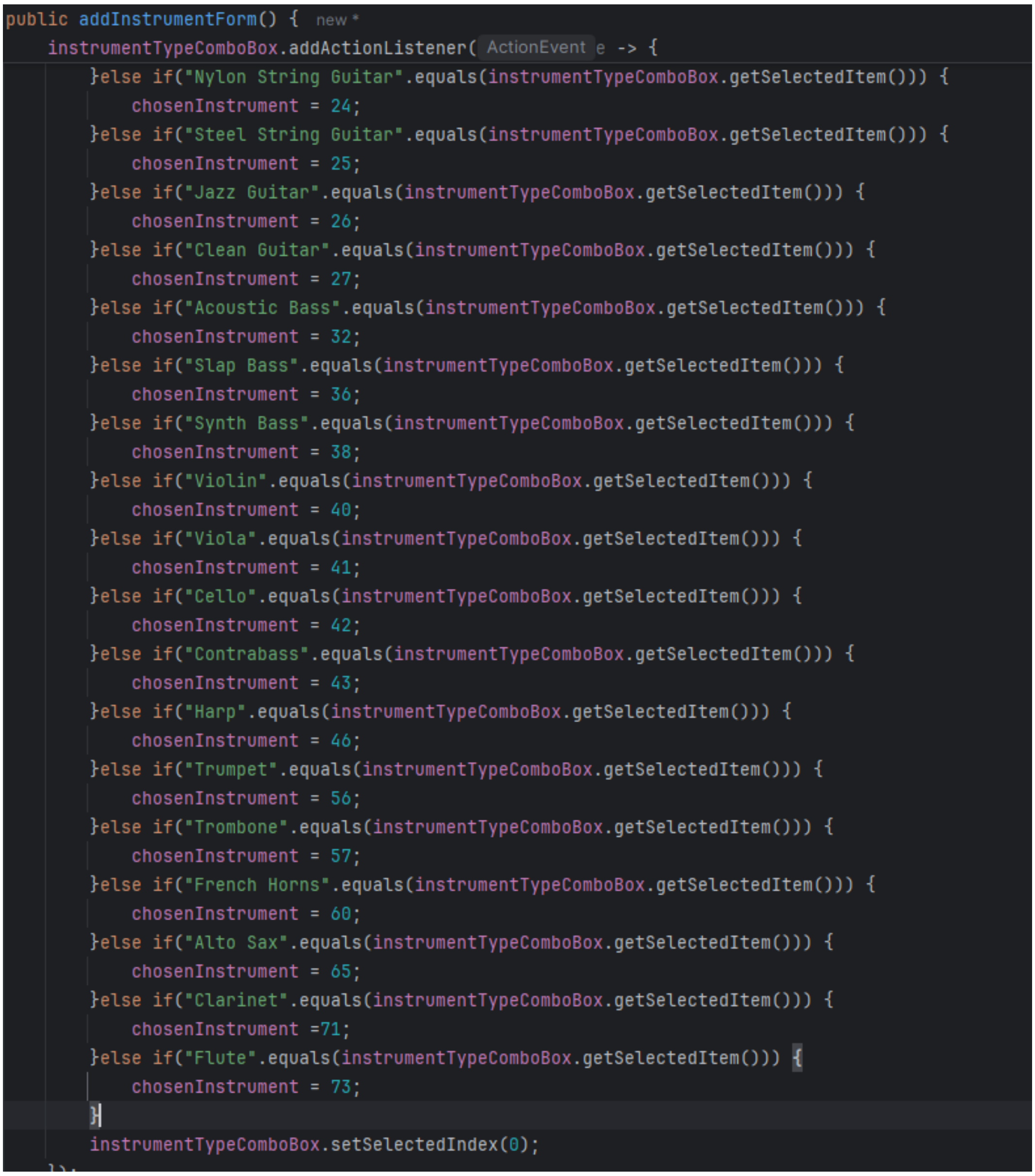
I then changed the model for the instrumentTypeComboBox so that it included various instruments that can be added to the synthesiser. I did this so that the user has a variety of different sounds to add to their sequence:



Items in the model (one line per item):

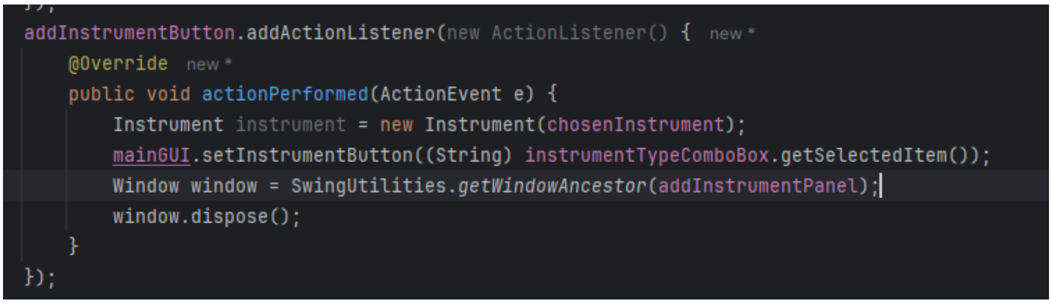
```
Acoustic Bass
Alto Sax
Cello
Clean Guitar
Clarinet
Contrabass
Electric Piano 1
Electric Piano 2
Flute
French Horns
Harp
Jazz Guitar
Nylon String Guitar
Piano
Reed Organ
Slap Bass
```

I then created an action listener for the instrumentTypeComboBox, which changes the value of chosenInstrument to a specific integer value (based on the user input) which relates to the index of the instrument in the javax.sound.mimd.Instrument[] array. This is so that the correct sound could be loaded into the synthesizer and sequencer:

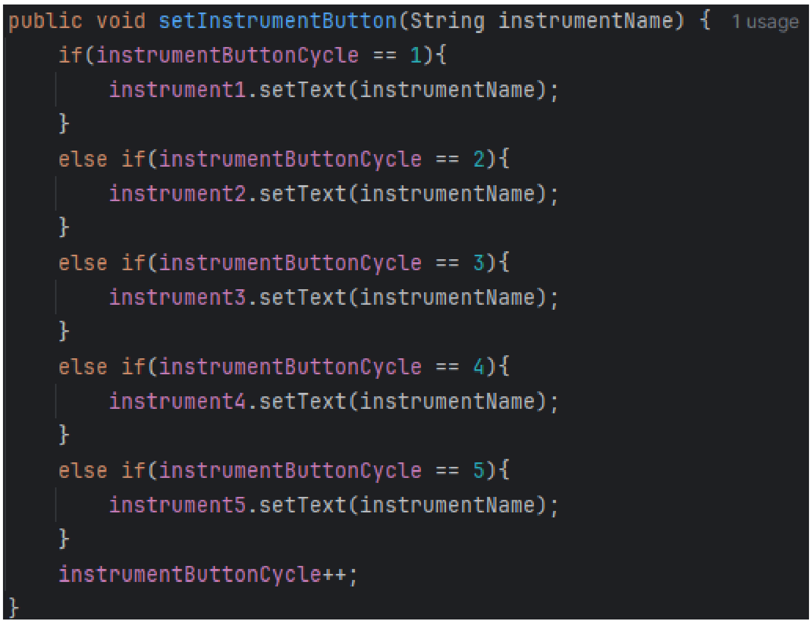


```
public addInstrumentForm() {  new *
    instrumentTypeComboBox.addActionListener( ActionEvent e -> {
        }else if("Nylon String Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 24;
        }else if("Steel String Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 25;
        }else if("Jazz Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 26;
        }else if("Clean Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 27;
        }else if("Acoustic Bass".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 32;
        }else if("Slap Bass".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 36;
        }else if("Synth Bass".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 38;
        }else if("Violin".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 40;
        }else if("Viola".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 41;
        }else if("Cello".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 42;
        }else if("Contrabass".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 43;
        }else if("Harp".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 46;
        }else if("Trumpet".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 56;
        }else if("Trombone".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 57;
        }else if("French Horns".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 60;
        }else if("Alto Sax".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 65;
        }else if("Clarinet".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument =71;
        }else if("Flute".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 73;
        }
        instrumentTypeComboBox.setSelectedIndex(0);
```

I then created another Action Listener for the addInstrumentButton, so that when the user clicks apply, the addInstrumentForm closes, and the instrument is added to the sequencer. This allows the user to add their desired instrument to the sequencer. Here is the code:

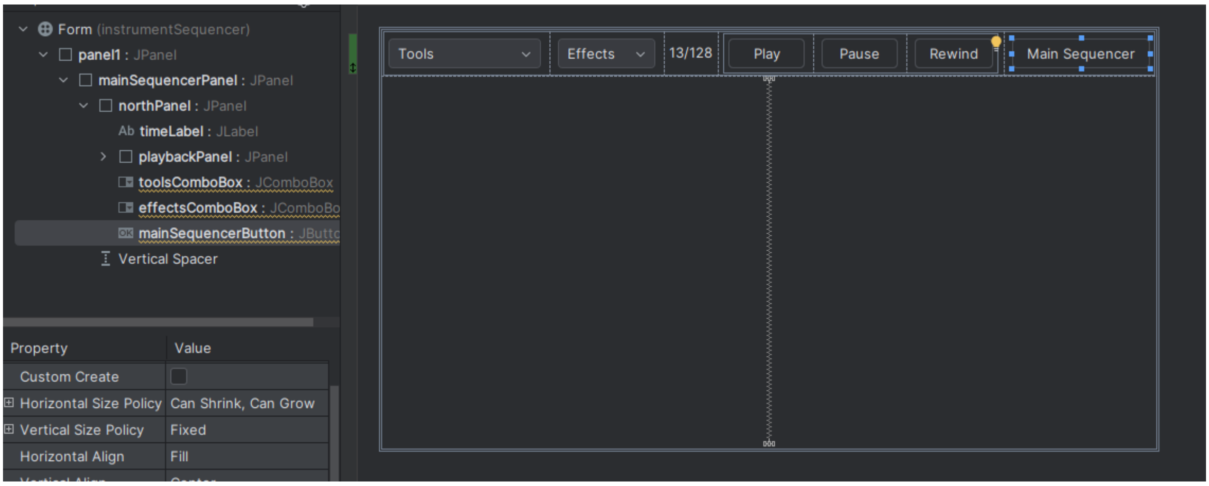


```
addInstrumentButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = new Instrument(chosenInstrument);
        mainGUI.setInstrumentButton((String) instrumentTypeComboBox.getSelectedItem());
        Window window = SwingUtilities.getWindowAncestor(addInstrumentPanel);
        window.dispose();
    }
});
```



```
public void setInstrumentButton(String instrumentName) {  1 usage
    if(instrumentButtonCycle == 1){
        instrument1.setText(instrumentName);
    }
    else if(instrumentButtonCycle == 2){
        instrument2.setText(instrumentName);
    }
    else if(instrumentButtonCycle == 3){
        instrument3.setText(instrumentName);
    }
    else if(instrumentButtonCycle == 4){
        instrument4.setText(instrumentName);
    }
    else if(instrumentButtonCycle == 5){
        instrument5.setText(instrumentName);
    }
    instrumentButtonCycle++;
}
```

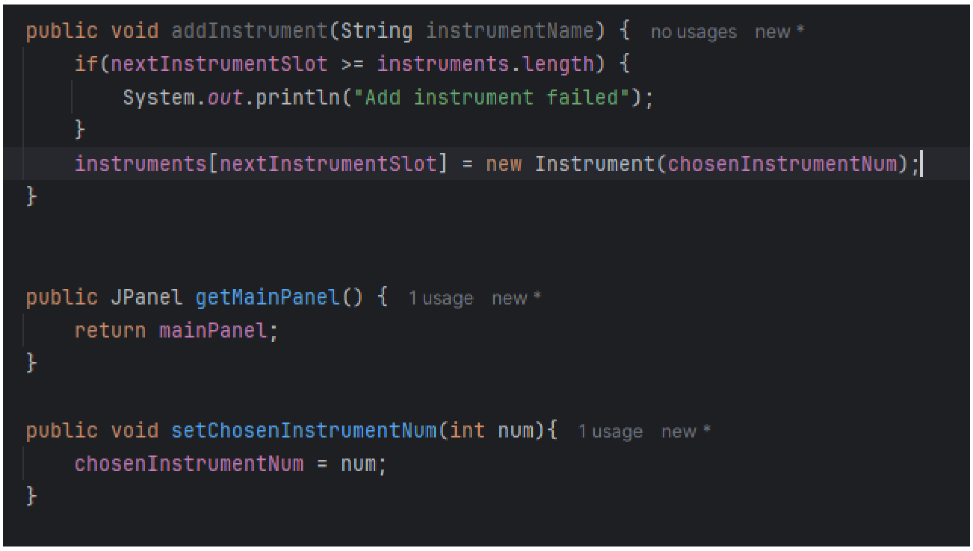
Next, I created another GUI form for the individual instrument sequencer called instrumentSequencer. I added Jbuttons, JComboBox's and a JLabel. The northPanel of this GUI is like that of the mainGUI, since they both contain sequencing and playback features. I created this GUI and the components within so that the user can sequence one instrument before combining it with the others to make music. Here is the instrumentSequencer GUI:



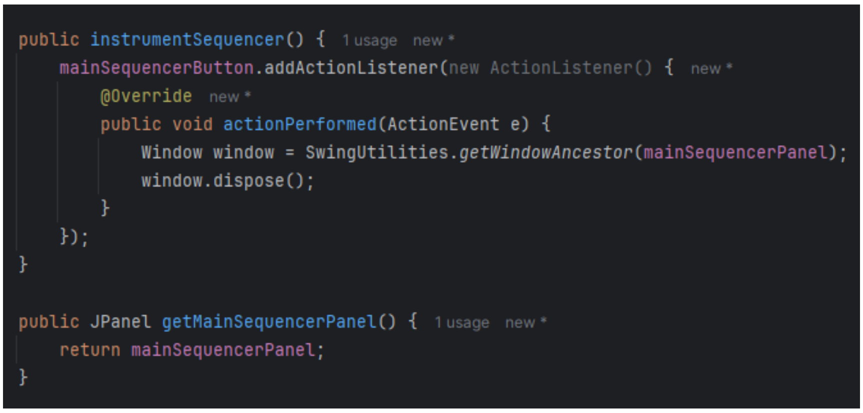
```
Form (instrumentSequencer)
  panel1 : JPanel
    mainSequencerPanel : JPanel
      northPanel : JPanel
        timeLabel : JLabel
        playbackPanel : JPanel
        toolsComboBox : JComboBox
        effectsComboBox : JComboBo
        mainSequencerButton : JButto
      Vertical Spacer
```

Property	Value
Custom Create	□
⊞ Horizontal Size Policy	Can Shrink, Can Grow
⊞ Vertical Size Policy	Fixed
Horizontal Align	Fill
Vertical Align	Center

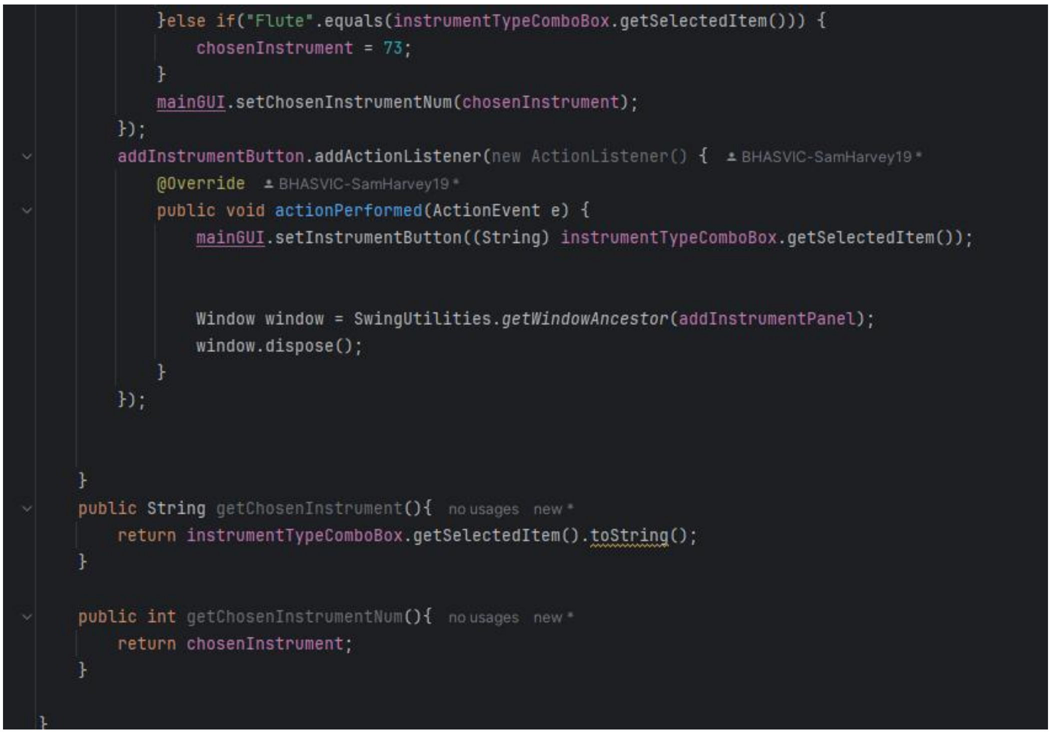
I also removed the tools and effects JComboBox from the GUI since they were not necessary for the user to use in this GUI. In addition, I added a JProgressBar so that the user can see how far along the playback they are, and a JScrollPane, which contains the sequencerSplitPlane, which further contains the pianoKeysPanel and sequencerTable. All these components will make up a sequencer which the user can add/remove notes to by clicking on the table. I also changed various things in previous code, such as the way that instruments are added/removed. I did this so that it easier to code the different instruments having their own different sequencers. This made me hold all the instruments in an array so that they can all be accessed through one array. Here is all the code that I changed/added:



```java
public void addInstrument(String instrumentName) {  no usages  new *
    if(nextInstrumentSlot >= instruments.length) {
        System.out.println("Add instrument failed");
    }
    instruments[nextInstrumentSlot] = new Instrument(chosenInstrumentNum);
}


public JPanel getMainPanel() {  1 usage  new *
    return mainPanel;
}

public void setChosenInstrumentNum(int num){  1 usage  new *
    chosenInstrumentNum = num;
}
```

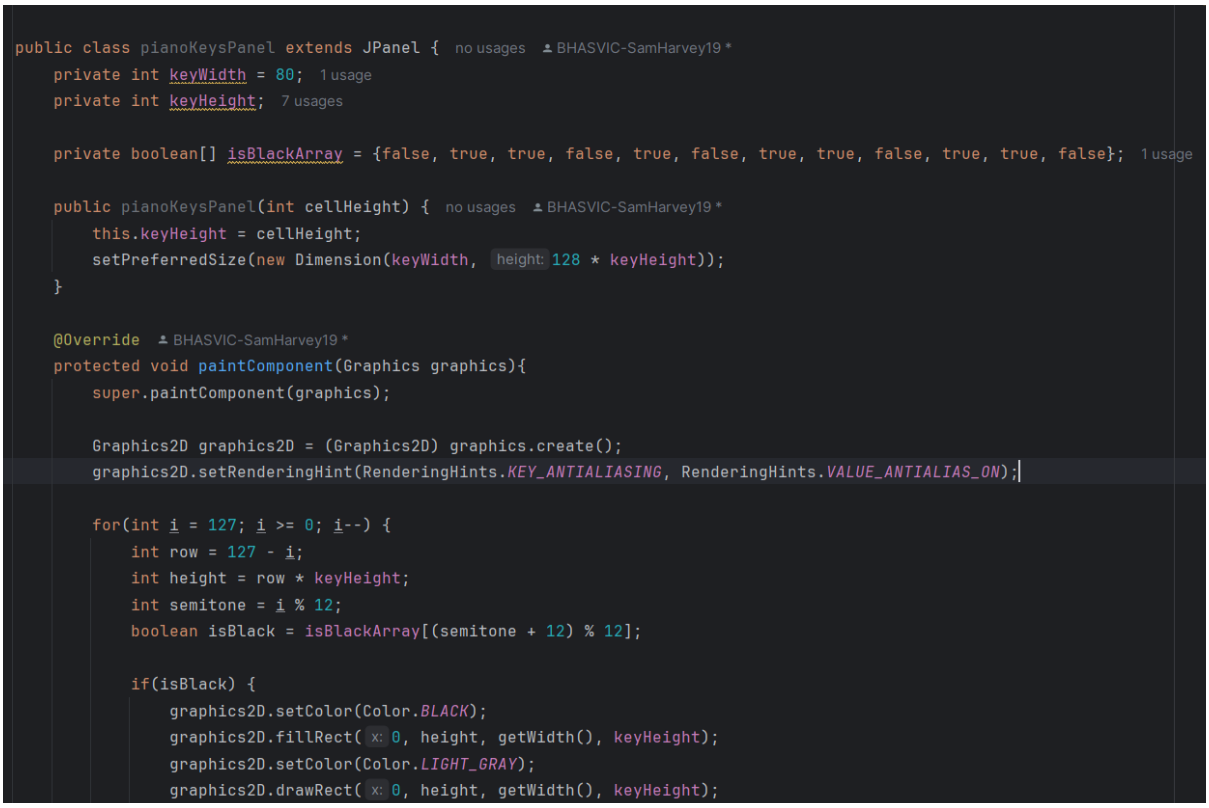


```java
public instrumentSequencer() {  1 usage  new *
    mainSequencerButton.addActionListener(new ActionListener() {  new *
        @Override  new *
        public void actionPerformed(ActionEvent e) {
            Window window = SwingUtilities.getWindowAncestor(mainSequencerPanel);
            window.dispose();
        }
    });
}

public JPanel getMainSequencerPanel() {  1 usage  new *
    return mainSequencerPanel;
}
```

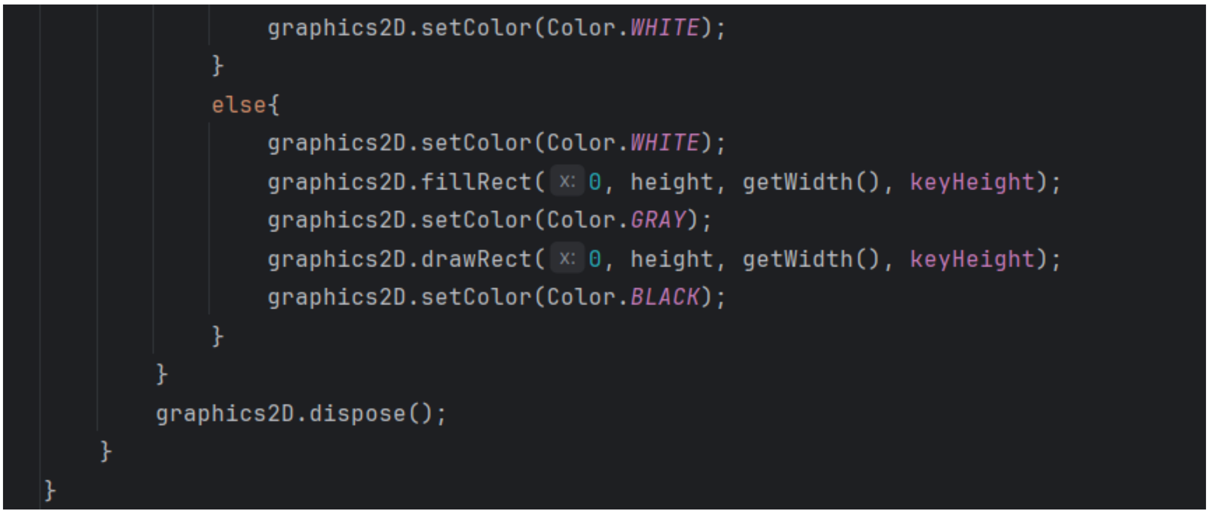


```java
                }else if("Flute".equals(instrumentTypeComboBox.getSelectedItem())) {
                    chosenInstrument = 73;
                }
                mainGUI.setChosenInstrumentNum(chosenInstrument);
            });
            addInstrumentButton.addActionListener(new ActionListener() {  BHASVIC-SamHarvey19 *
                @Override  BHASVIC-SamHarvey19 *
                public void actionPerformed(ActionEvent e) {
                    mainGUI.setInstrumentButton((String) instrumentTypeComboBox.getSelectedItem());


                    Window window = SwingUtilities.getWindowAncestor(addInstrumentPanel);
                    window.dispose();
                }
            });


        }
        public String getChosenInstrument(){  no usages  new *
            return instrumentTypeComboBox.getSelectedItem().toString();
        }

        public int getChosenInstrumentNum(){  no usages  new *
            return chosenInstrument;
        }

    }
```

Next, I created a class for the pianoKeysPanel, so that I could display the piano keys on the left side of the individual instrument sequencer. I did this so that the user knows what notes they are adding into their sequence. To help me with this module, I referred to bogotobogo (https://www.bogotobogo.com/Java/tutorials/javagraphics3.php), which taught me how to use java's graphics interface, and paintComponent(). The reason I used this is because the piano does not have to be interactive, but only needs to display the keys, meaning that I can simply paint them onto the pianoKeysPanel. This is the code that I made to display the piano keys:

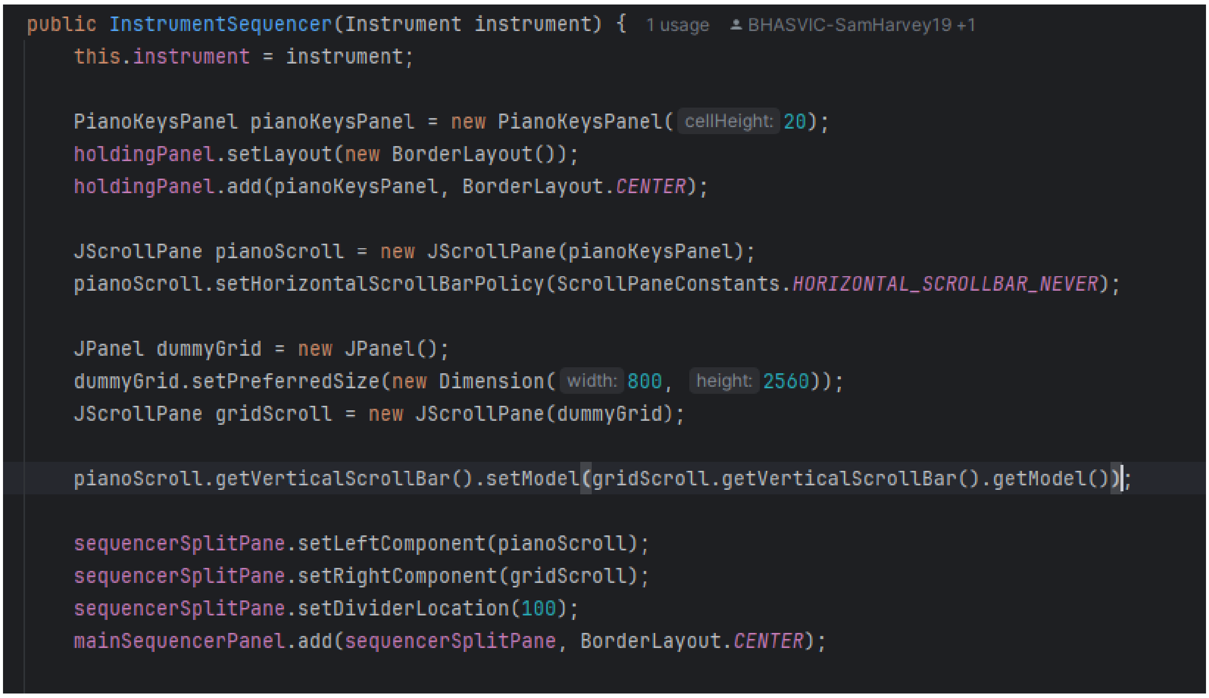


```
public class pianoKeysPanel extends JPanel {  no usages  ≗ BHASVIC-SamHarvey19 *
    private int keyWidth = 80;  1 usage
    private int keyHeight;  7 usages

    private boolean[] isBlackArray = {false, true, true, false, true, false, true, true, false, true, true, false};  1 usage

    public pianoKeysPanel(int cellHeight) {  no usages  ≗ BHASVIC-SamHarvey19 *
        this.keyHeight = cellHeight;
        setPreferredSize(new Dimension(keyWidth,  height: 128 * keyHeight));
    }

    @Override  ≗ BHASVIC-SamHarvey19 *
    protected void paintComponent(Graphics graphics){
        super.paintComponent(graphics);

        Graphics2D graphics2D = (Graphics2D) graphics.create();
        graphics2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        for(int i = 127; i >= 0; i--) {
            int row = 127 - i;
            int height = row * keyHeight;
            int semitone = i % 12;
            boolean isBlack = isBlackArray[(semitone + 12) % 12];

            if(isBlack) {
                graphics2D.setColor(Color.BLACK);
                graphics2D.fillRect( x: 0, height, getWidth(), keyHeight);
                graphics2D.setColor(Color.LIGHT_GRAY);
                graphics2D.drawRect( x: 0, height, getWidth(), keyHeight);
```



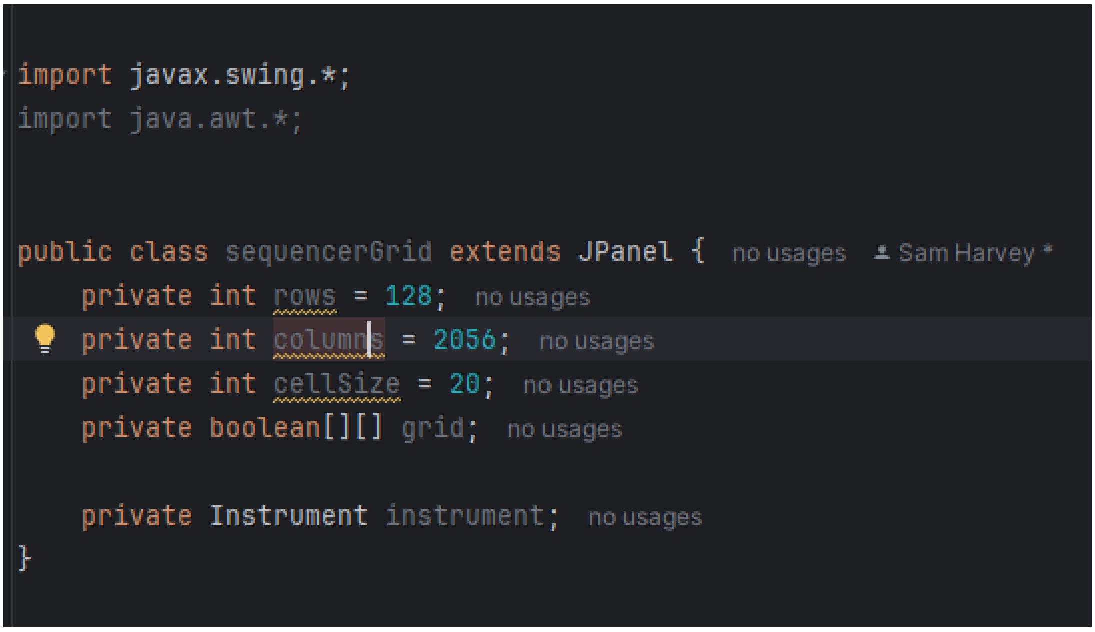
```
                graphics2D.setColor(Color.WHITE);
            }
            else{
                graphics2D.setColor(Color.WHITE);
                graphics2D.fillRect( x: 0, height, getWidth(), keyHeight);
                graphics2D.setColor(Color.GRAY);
                graphics2D.drawRect( x: 0, height, getWidth(), keyHeight);
                graphics2D.setColor(Color.BLACK);
            }
        }
        graphics2D.dispose();
    }
}
```

Next, I had to make this component visible on the instrumentSequencer, so that the piano could be seen and utilised by the user. I also needed to make the sequencer able to scroll, so I used geeksforgeeks (https://www.geeksforgeeks.org/java/java-jscrollpane/) to help me with this. The reason that this was necessary is because the user must be able to view all the notes that are available to be added to the sequence. I also used this website to help me to sync the scroll bars for the pianoScroll and gridScroll. I did this so that all the midi notes on the grid and the piano visual would line up correctly when the sequencerGrid is added. Just to note, the dummyGrid in this module of code is not actually going to be in the final code, since it is just a placeholder for the sequencerGrid that I will make soon. I added this dummyGrid so I could test if the sequencerSplitPane correctly adds the components to the correct areas. Here is the code:



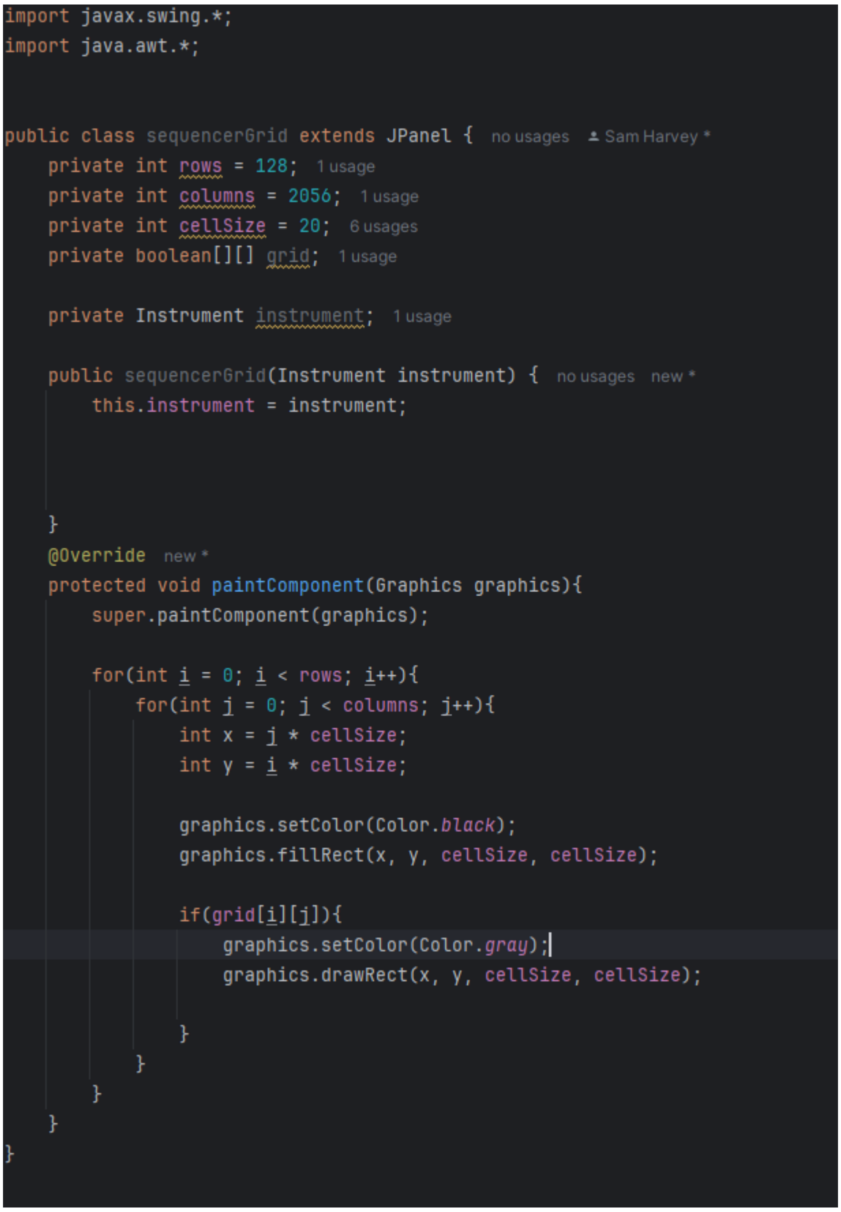
```
public InstrumentSequencer(Instrument instrument) {  1 usage   BHASVIC-SamHarvey19 +1
    this.instrument = instrument;

    PianoKeysPanel pianoKeysPanel = new PianoKeysPanel( cellHeight: 20);
    holdingPanel.setLayout(new BorderLayout());
    holdingPanel.add(pianoKeysPanel, BorderLayout.CENTER);

    JScrollPane pianoScroll = new JScrollPane(pianoKeysPanel);
    pianoScroll.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    JPanel dummyGrid = new JPanel();
    dummyGrid.setPreferredSize(new Dimension( width: 800,  height: 2560));
    JScrollPane gridScroll = new JScrollPane(dummyGrid);

    pianoScroll.getVerticalScrollBar().setModel(gridScroll.getVerticalScrollBar().getModel());

    sequencerSplitPane.setLeftComponent(pianoScroll);
    sequencerSplitPane.setRightComponent(gridScroll);
    sequencerSplitPane.setDividerLocation(100);
    mainSequencerPanel.add(sequencerSplitPane, BorderLayout.CENTER);
```

After that was tested and made sure it full worked, I started to code the sequencerGrid so that notes were able to be added into the sequence. Firstly, I added all the variables that I would need for this class. “rows” referrers to the pitches of each midi note, “columns” referrers to the number of steps that are in the timeline, “cellSize” is the number of pixels per each cell on the grid, “grid[][]” referrers to the on/off states of each note. I also added an instrument as a private attribute, so that I could add notes to a specific instrument when a cell is clicked. Here is the code:



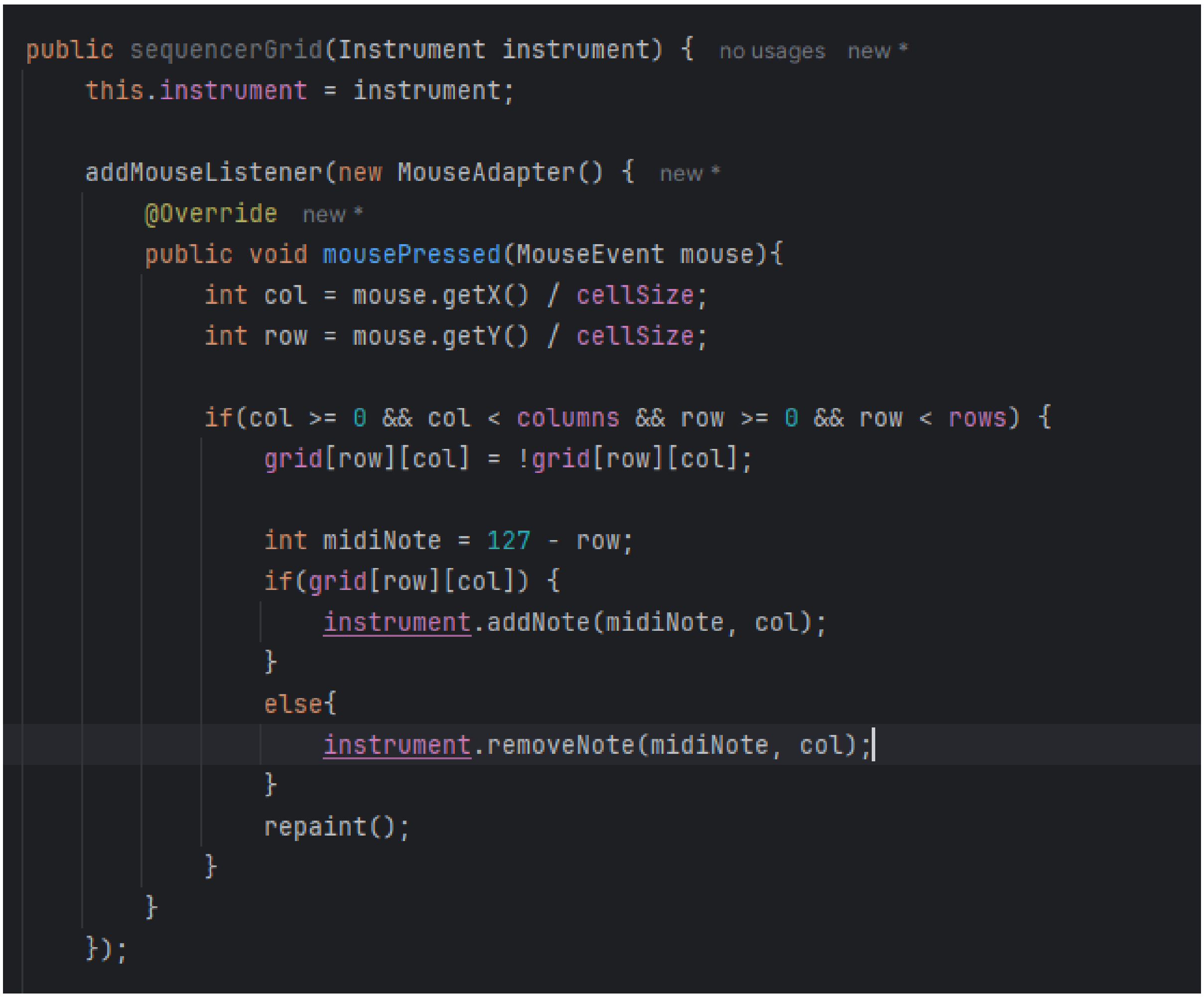
```
import javax.swing.*;
import java.awt.*;


public class sequencerGrid extends JPanel {  no usages  Sam Harvey *
    private int rows = 128;  no usages
    private int columns = 2056;  no usages
    private int cellSize = 20;  no usages
    private boolean[][] grid;  no usages

    private Instrument instrument;  no usages
}
```

After this was done, I added the constructor for the sequencerGrid so that each cell could be painted on. I again used bogotobogo (https://www.bogotobogo.com/Java/tutorials/javagraphics3.php) to remind me how to use paintComponent and other related features. The paintComponent method is used to add a grid to the sequencerGrid, and fill a cell if it has been clicked. This is where I will use the grid[][] attribute to check if it has been clicked and fill it if it has. This creates a good interface where the user can add/remove sounds from each of their instruments. This is the code:



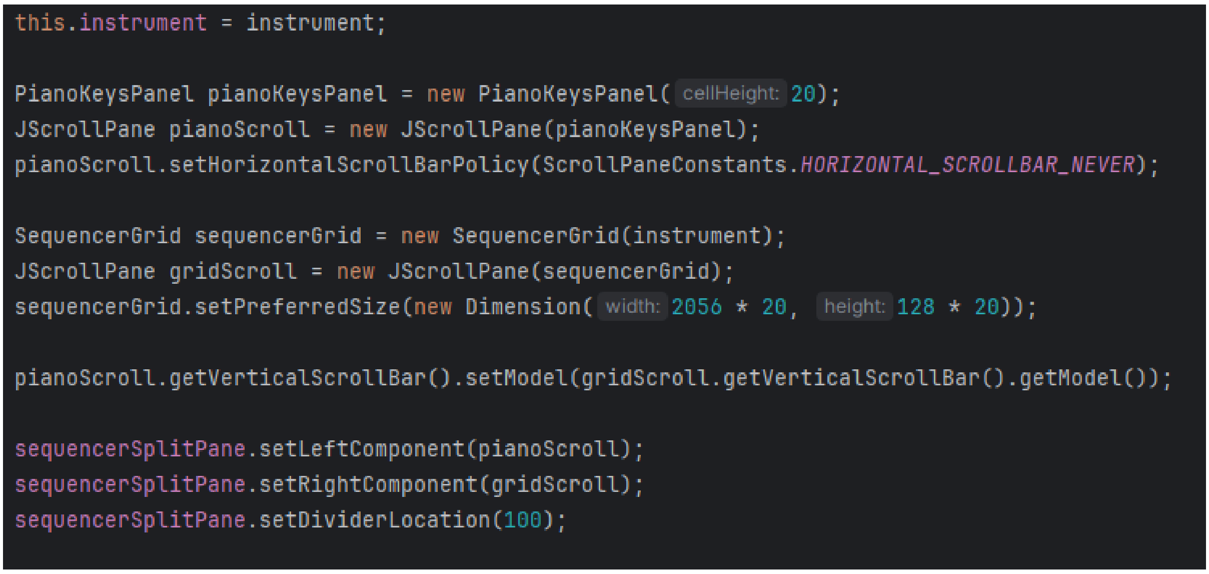
```
import javax.swing.*;
import java.awt.*;


public class sequencerGrid extends JPanel {  no usages  Sam Harvey *
    private int rows = 128;  1 usage
    private int columns = 2056;  1 usage
    private int cellSize = 20;  6 usages
    private boolean[][] grid;  1 usage

    private Instrument instrument;  1 usage

    public sequencerGrid(Instrument instrument) {  no usages  new *
        this.instrument = instrument;



    }
    @Override  new *
    protected void paintComponent(Graphics graphics){
        super.paintComponent(graphics);

        for(int i = 0; i < rows; i++){
            for(int j = 0; j < columns; j++){
                int x = j * cellSize;
                int y = i * cellSize;

                graphics.setColor(Color.black);
                graphics.fillRect(x, y, cellSize, cellSize);

                if(grid[i][j]){
                    graphics.setColor(Color.gray);
                    graphics.drawRect(x, y, cellSize, cellSize);

                }
            }
        }
    }
}
```

I then added a mouseListenener to the sequencerGrid class so that it could detect when the mouse is pressed on the grid and then fill that box and run the addNote method in the instrument class (if the note is off) to add a midi note to the sequence or the removeNote method (if the note is on). This is the main feature of my entire program, since it is essential for the user to be able to add/remove notes though this mechanism. This is the code that I created for this:



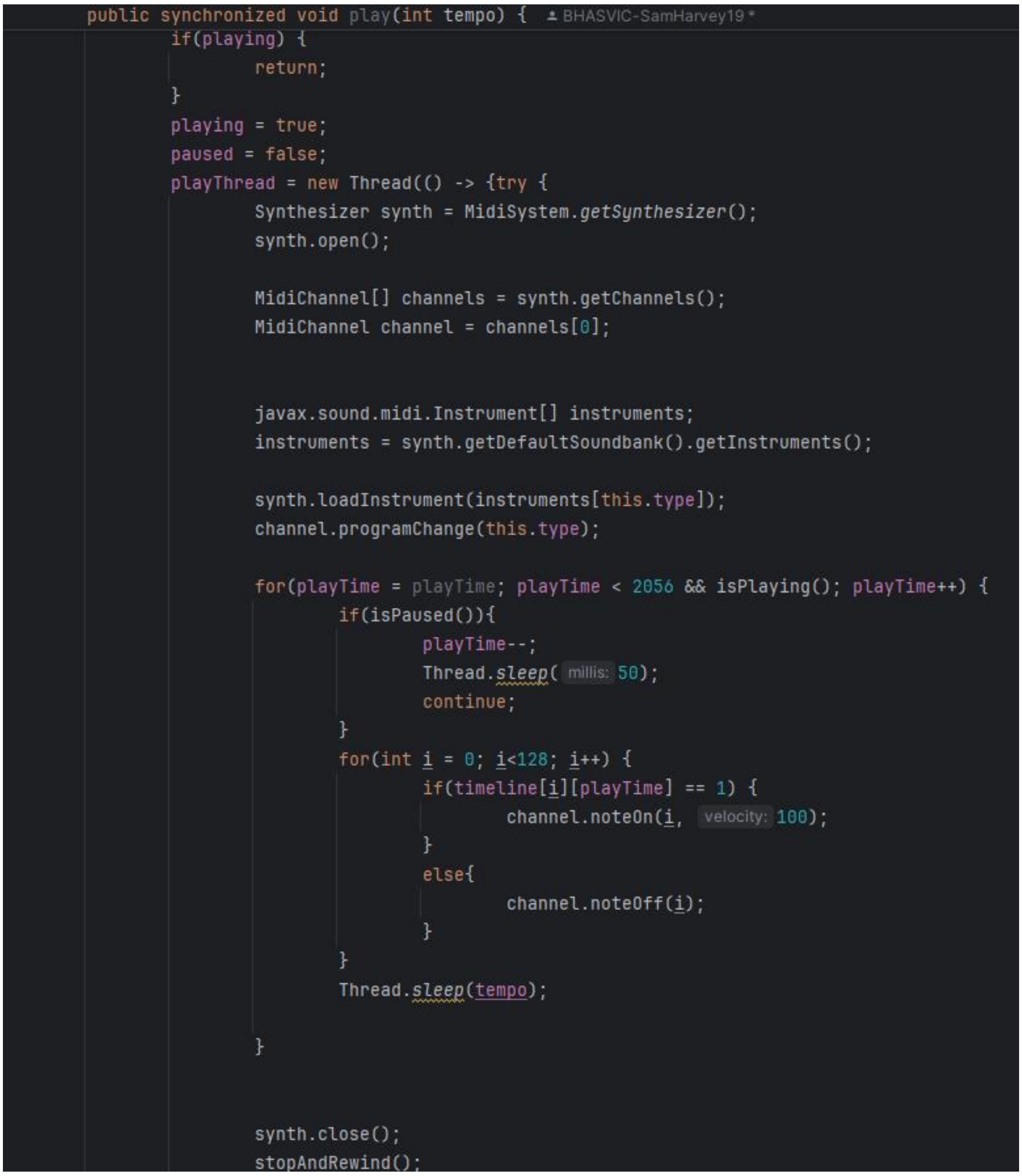
```
public sequencerGrid(Instrument instrument) {  no usages  new *
    this.instrument = instrument;

    addMouseListener(new MouseAdapter() {  new *
        @Override  new *
        public void mousePressed(MouseEvent mouse){
            int col = mouse.getX() / cellSize;
            int row = mouse.getY() / cellSize;

            if(col >= 0 && col < columns && row >= 0 && row < rows) {
                grid[row][col] = !grid[row][col];

                int midiNote = 127 - row;
                if(grid[row][col]) {
                    instrument.addNote(midiNote, col);
                }
                else{
                    instrument.removeNote(midiNote, col);
                }
                repaint();
            }
        }
    });
```

The purpose of this code is to detect when the mouse is pressed and then set the “col” and “row” to (where the mouse was pressed)/cellSize. This means that the mouse’s location when pressed will relate to a cell on the grid. The if statement simply checks that the mouse’s location when pressed is in suitable bounds, before changing the state of the grid[][] attribute, and then turning a note on (if it was previously off) or off if it was previously on). I then called the repaint() method so that the grid could be changed in relation with this click. The next step was to add this sequencerGrid in place for the dummyGrid that was previously added to the instrumentSequencer GUI. This was the code that did that:

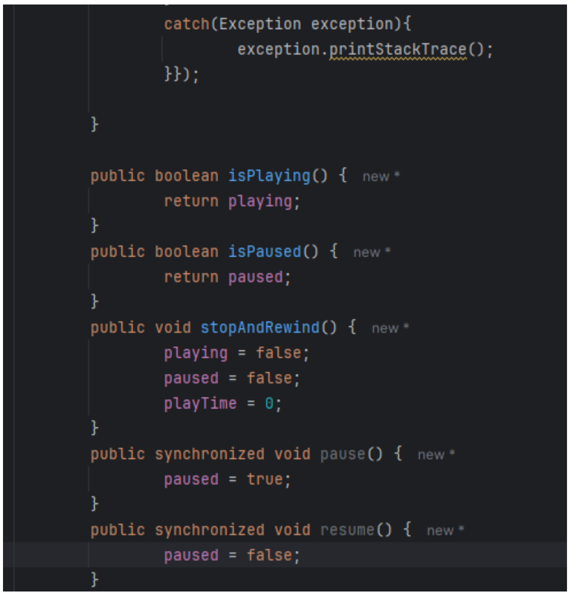


```
this.instrument = instrument;

PianoKeysPanel pianoKeysPanel = new PianoKeysPanel( cellHeight: 20);
JScrollPane pianoScroll = new JScrollPane(pianoKeysPanel);
pianoScroll.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

SequencerGrid sequencerGrid = new SequencerGrid(instrument);
JScrollPane gridScroll = new JScrollPane(sequencerGrid);
sequencerGrid.setPreferredSize(new Dimension( width: 2056 * 20,  height: 128 * 20));

pianoScroll.getVerticalScrollBar().setModel(gridScroll.getVerticalScrollBar().getModel());

sequencerSplitPane.setLeftComponent(pianoScroll);
sequencerSplitPane.setRightComponent(gridScroll);
sequencerSplitPane.setDividerLocation(100);
```

When I tested this, I realised how large the piano keys and grid were, so I changed the size of each cell on the grid to 10, and the height of each key on the piano visual to 10 also. This meant that the sequencer will be smaller and therefore more user friendly, since there is less scrolling required to get to higher/lower notes. The next part that I added was linking the play, pause and rewind button in both the individual instrument sequencer and the mainGUI to the play() method in the Instrument class. I started by making necessary changes to the play() method in the instrument class, and adding other methods to make the play, pause and rewind features work properly. This is an important feature of my program, so I spent a lot of time perfecting it, and learning things that I must know to do it. This includes visiting geeksforgeeks tutorial on threads (https://www.geeksforgeeks.org/java/java-multithreading-tutorial/), so that I was able to synchronise threads and utilise them in all ways that I need to. Here is the code:

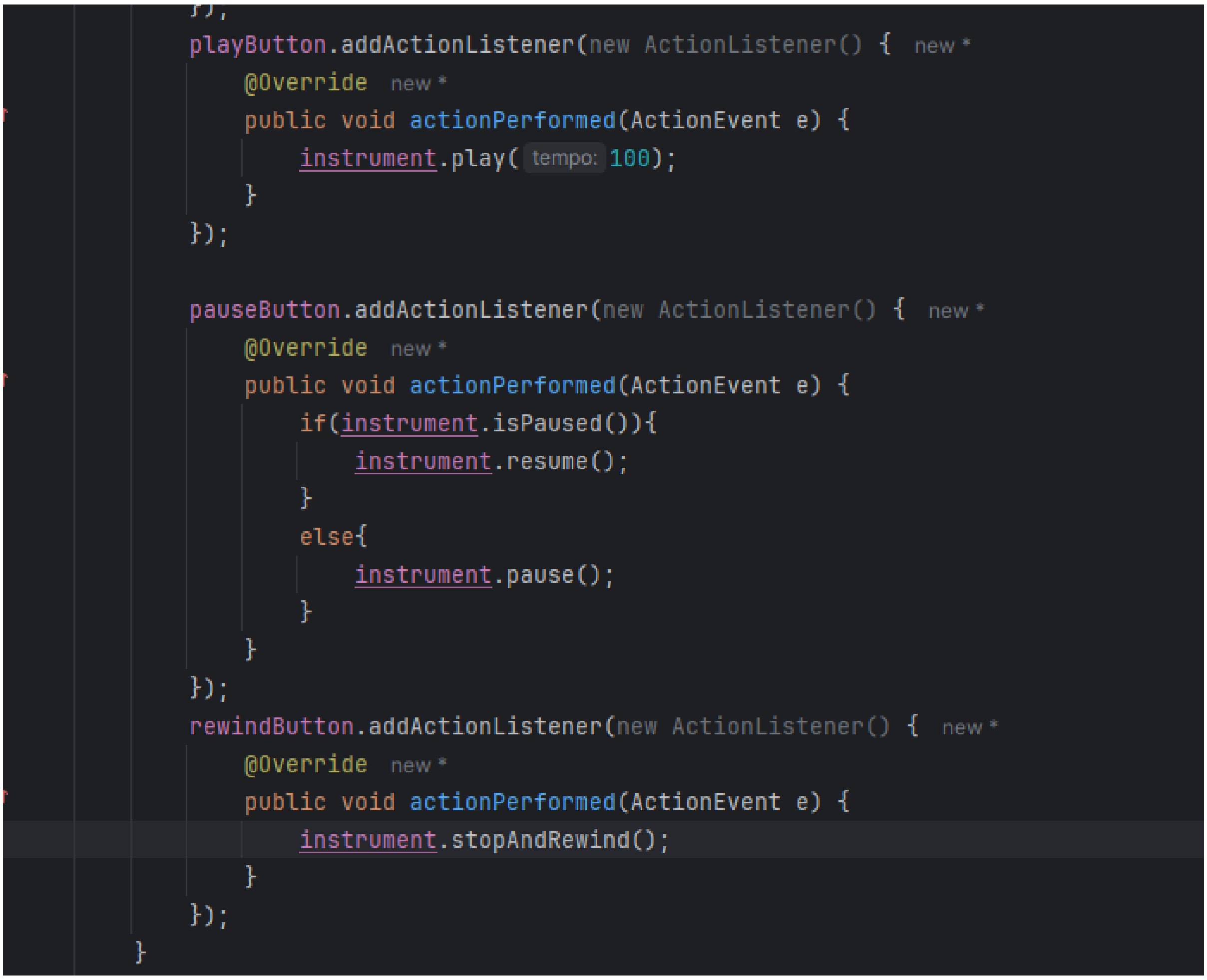


```java
public synchronized void play(int tempo) {  BHASVIC-SamHarvey19 *
    if(playing) {
        return;
    }
    playing = true;
    paused = false;
    playThread = new Thread(() -> {try {
        Synthesizer synth = MidiSystem.getSynthesizer();
        synth.open();

        MidiChannel[] channels = synth.getChannels();
        MidiChannel channel = channels[0];


        javax.sound.midi.Instrument[] instruments;
        instruments = synth.getDefaultSoundbank().getInstruments();

        synth.loadInstrument(instruments[this.type]);
        channel.programChange(this.type);

        for(playTime = playTime; playTime < 2056 && isPlaying(); playTime++) {
            if(isPaused()){
                playTime--;
                Thread.sleep( millis: 50);
                continue;
            }
            for(int i = 0; i<128; i++) {
                if(timeline[i][playTime] == 1) {
                    channel.noteOn(i,  velocity: 100);
                }
                else{
                    channel.noteOff(i);
                }
            }
            Thread.sleep(tempo);

        }


        synth.close();
        stopAndRewind();
```



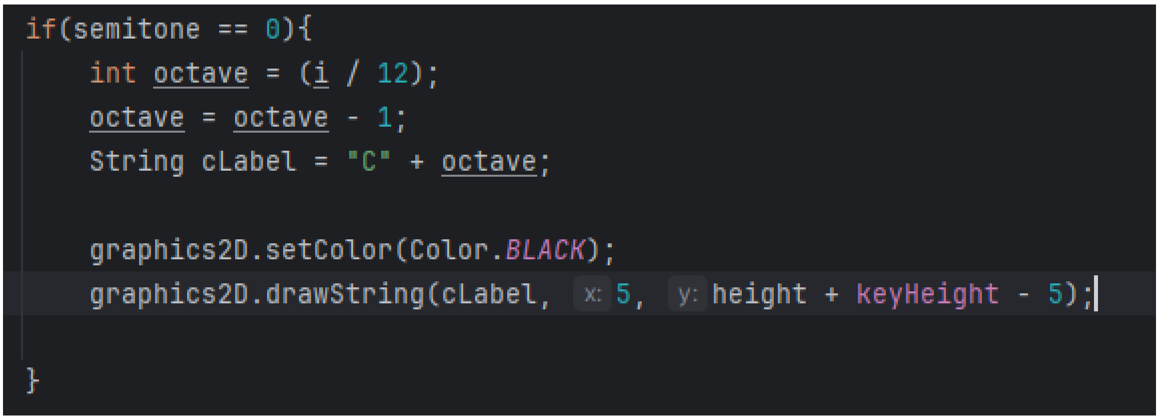
```java
        catch(Exception exception){
            exception.printStackTrace();
        }});

}

public boolean isPlaying() {  new *
        return playing;
}
public boolean isPaused() {  new *
        return paused;
}
public void stopAndRewind() {  new *
        playing = false;
        paused = false;
        playTime = 0;
}
public synchronized void pause() {  new *
        paused = true;
}
public synchronized void resume() {  new *
        paused = false;
}
```

This is the instruments play() method. As you can see, I had to change various parts of it so it would be synchronized with the playTime thread. This allows the user to pause, resume, and rewind the music whenever they desire. This is therefore important as it will make my program more user-friendly and give more features to the user. The next thing I did was create action listeners for the play, pause and rewind buttons within the instrumentSequencer GUI. Inside these action listeners, there are small blocks of code which are relevant for running the play(), pause(), resume(), or stopAndRewind() methods from the instrument class. This is the code:



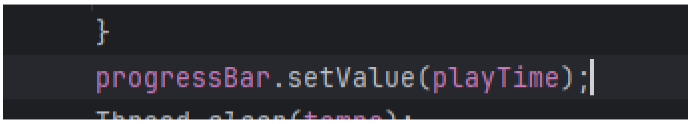
```
        });
        playButton.addActionListener(new ActionListener() {  new *
            @Override  new *
            public void actionPerformed(ActionEvent e) {
                instrument.play( tempo: 100);
            }
        });

        pauseButton.addActionListener(new ActionListener() {  new *
            @Override  new *
            public void actionPerformed(ActionEvent e) {
                if(instrument.isPaused()){
                    instrument.resume();
                }
                else{
                    instrument.pause();
                }
            }
        });
        rewindButton.addActionListener(new ActionListener() {  new *
            @Override  new *
            public void actionPerformed(ActionEvent e) {
                instrument.stopAndRewind();
            }
        });
    }
```

In the actual code, the tempo parameter for the play() method will not always be 100, but instead will be passed in, so that the user can state what tempo they would like the music to play at. I tested this by adding various random notes into a sequence on different instruments and it worked. The next step was to make a bar that will travel across the grid as the music is played. This will allow the user to see where in the music is currently being played, and therefore gives them a more user-friendly environment to make music in. Before I did this however, I realised that it is hard to see where you are on the piano notes when adding notes to the sequence. To fix this, I added a label on every C note (C1, C2, C3) respectively, so that the user can see the exact pitch of the notes that they are adding. To help me with this, I again referred to bogotobogo (https://www.bogotobogo.com/Java/tutorials/javagraphics3.php). Here is the code that I added to the paintComponent() method in the pianoKeysPanel class:

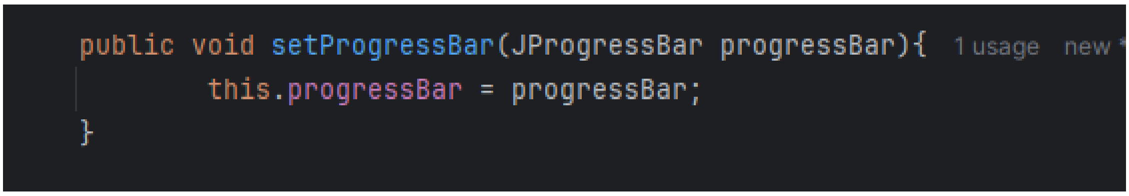


```
if(semitone == 0){
    int octave = (i / 12);
    octave = octave - 1;
    String cLabel = "C" + octave;

    graphics2D.setColor(Color.BLACK);
    graphics2D.drawString(cLabel,  x: 5,  y: height + keyHeight - 5);

}
```

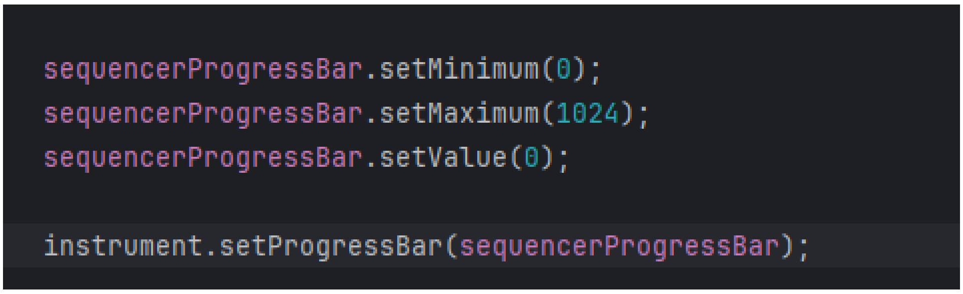
After that was completed, I set my focus back onto creating a moving bar that travels across the sequence as it is played. For this, I had to learn various methods that involve using JProgressBars (E.g. setValue(), setMinimum, setMaximum()). To learn this, I again referred to geeksforgeeks (https://www.geeksforgeeks.org/java/java-swing-jprogressbar/). This allowed me to create a method to set the sequencerProgressBar as a separate object within the Instrument class, so that I could update it every time that a beat is played in the sequence. I also added some code to make the progress bar set back to 0 when the music is stopped and rewind. Here is all the code that I added for this section:



```
    }
    progressBar.setValue(playTime);
```

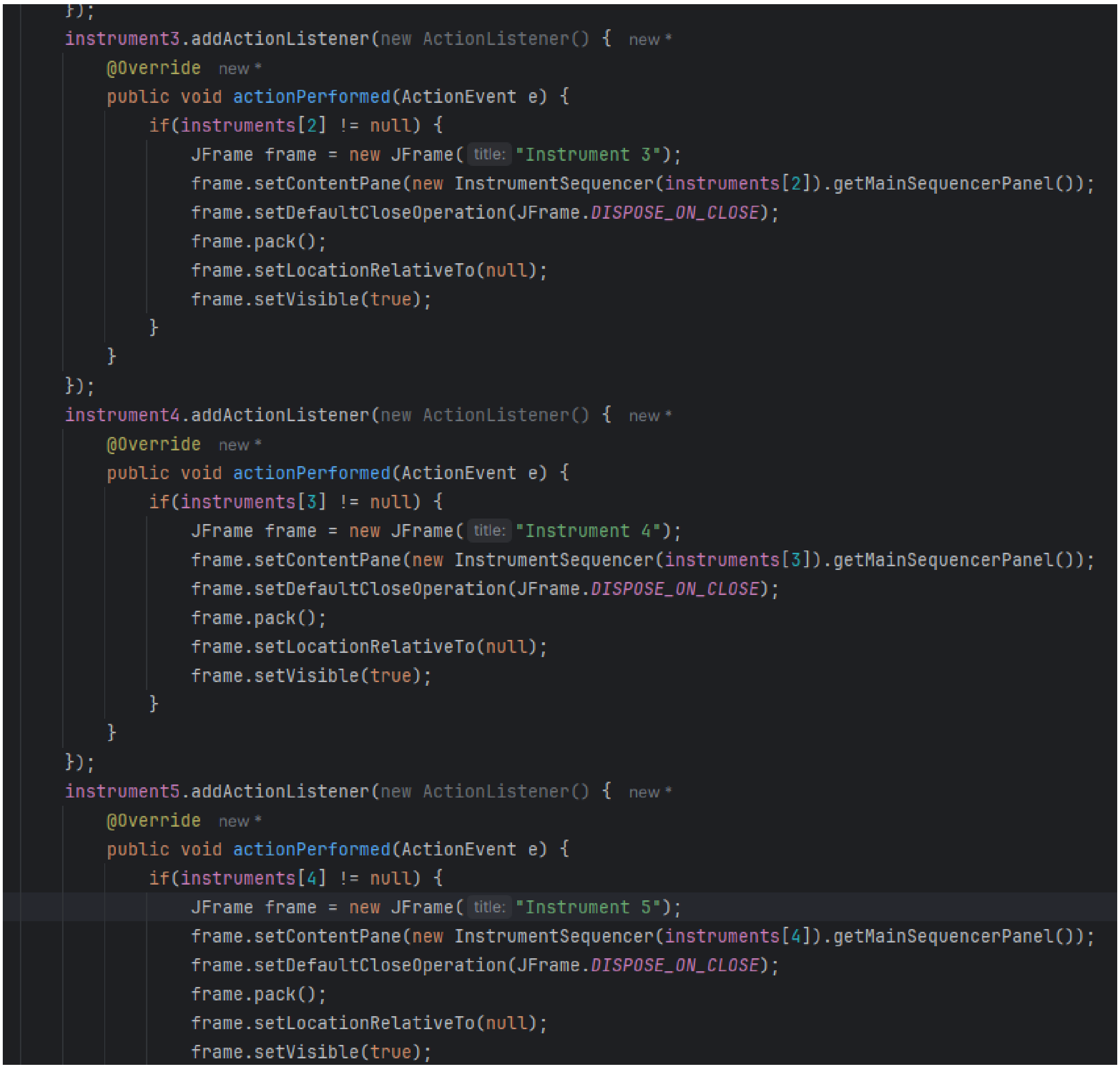


```
public void setProgressBar(JProgressBar progressBar){  1 usage  new *
        this.progressBar = progressBar;
}
```



```
sequencerProgressBar.setMinimum(0);
sequencerProgressBar.setMaximum(1024);
sequencerProgressBar.setValue(0);

instrument.setProgressBar(sequencerProgressBar);
```

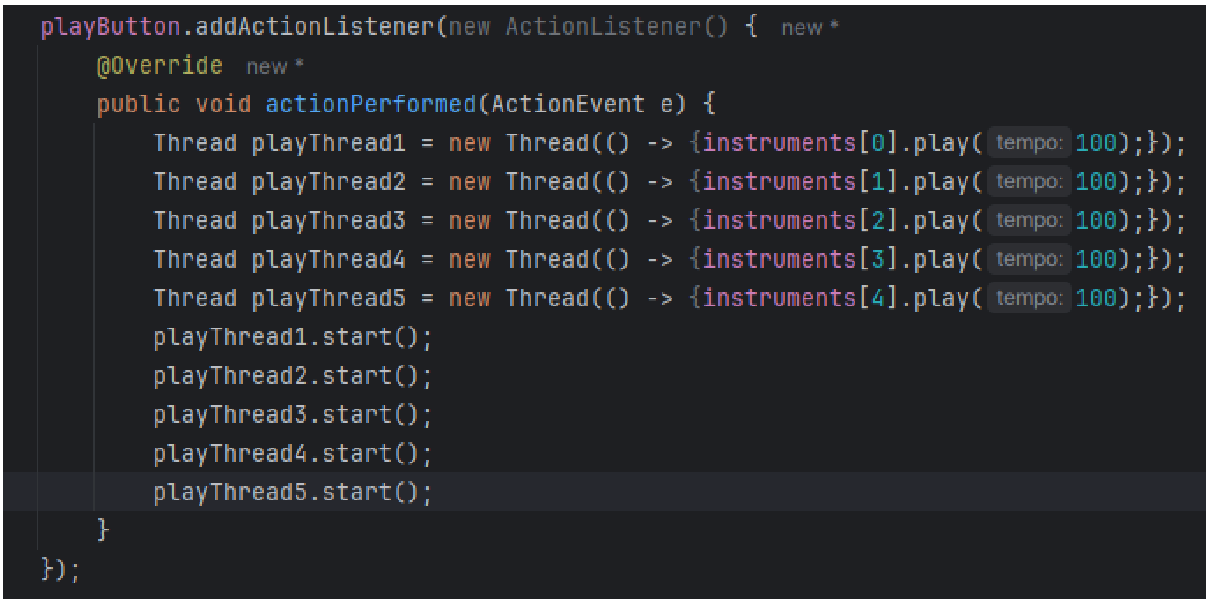
I tested this, and although the bar was slightly stuttered, it worked as expected. I expect this stutter was due to the power of the computer that I was coding on, so I did not see this as a problem. I am almost done with coding stage 1, however I did need to make it so that more than 1 instrument was able to be added. This involves me making Action listeners for instrument2, instrument3, instrument4, and instrument5 (all buttons), so that the user can click on them to access that instrument's sequencer. This is useful for the user, since it means that they can add multiple instruments to their sequence, and therefore create more complex and better music with my program. Here is the code:



```
    });
    instrument3.addActionListener(new ActionListener() {  new *
        @Override  new *
        public void actionPerformed(ActionEvent e) {
            if(instruments[2] != null) {
                JFrame frame = new JFrame( title: "Instrument 3");
                frame.setContentPane(new InstrumentSequencer(instruments[2]).getMainSequencerPanel());
                frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setVisible(true);
            }
        }
    });
    instrument4.addActionListener(new ActionListener() {  new *
        @Override  new *
        public void actionPerformed(ActionEvent e) {
            if(instruments[3] != null) {
                JFrame frame = new JFrame( title: "Instrument 4");
                frame.setContentPane(new InstrumentSequencer(instruments[3]).getMainSequencerPanel());
                frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setVisible(true);
            }
        }
    });
    instrument5.addActionListener(new ActionListener() {  new *
        @Override  new *
        public void actionPerformed(ActionEvent e) {
            if(instruments[4] != null) {
                JFrame frame = new JFrame( title: "Instrument 5");
                frame.setContentPane(new InstrumentSequencer(instruments[4]).getMainSequencerPanel());
                frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                frame.pack();
                frame.setLocationRelativeTo(null);
                frame.setVisible(true);
```

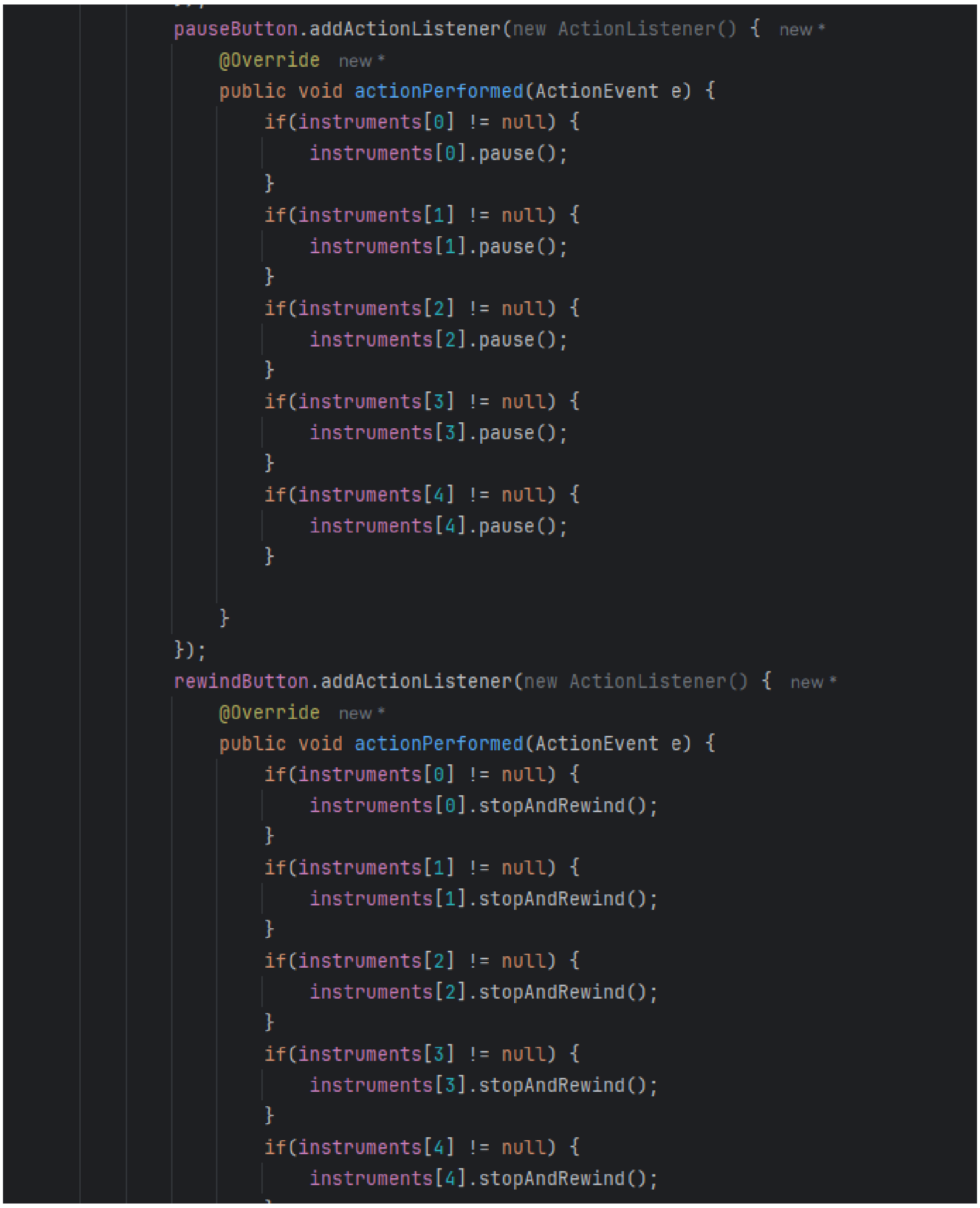
Next, I had to make the instrument1Timeline be a visual of the notes that are actually sequenced on that instrument. I did this not only for instrument1Timeline, but for all instrumentTimelines that are on the mainGUI. I did this by setting the contents of each of these JPanels to that of the SequencerGrid for each instrument. This means that the user can view what they have written for each instrument, without having to click on the instrument's button and access the instrumentSequencer. To do this, I used the paintComponent method again, so I again reminded myself how to do this by visiting bogotobogo (https://www.bogotobogo.com/Java/tutorials/javagraphics3.php). This allowed me to use this method to paint a much smaller version of the sequencerGrid onto the instrument1Timeline. This process involved me making a new JComponent called “painter”, which is painted on, and then added to the instrument1Timeline JPanel, so that the grid can be seen.

During the programming of this part, I spent a lot of time attempting and failing to do this with different methods. This is when I decided that this feature is not necessary and can be dispensed. This meant that I was able to move on to the last part of stage 1: making all instruments able to play at the same time, via the play button on the instrument sequencer. This will be done using multithreading, so I referenced geeksforgeeks (https://www.geeksforgeeks.org/java/multithreading-in-java/) again to help me with this. To start this module, I created an action listener for the play button on the mainGUI. This allowed me to detect when the user clicks the button, so I can run the necessary threads. This is the code for this segment:



```
playButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Thread playThread1 = new Thread(() -> {instruments[0].play( tempo: 100);});
        Thread playThread2 = new Thread(() -> {instruments[1].play( tempo: 100);});
        Thread playThread3 = new Thread(() -> {instruments[2].play( tempo: 100);});
        Thread playThread4 = new Thread(() -> {instruments[3].play( tempo: 100);});
        Thread playThread5 = new Thread(() -> {instruments[4].play( tempo: 100);});
        playThread1.start();
        playThread2.start();
        playThread3.start();
        playThread4.start();
        playThread5.start();
    }
});
```

I also needed an action listener for the stop and rewind buttons; however, this was simple since within the action listener I only needed to call the stopAndRewind method for rewind, or the pause method for pause. This allowed the user to have full availability for playback, making my program totally user friendly in this sense. This is the code for that small module:



```
pauseButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        if(instruments[0] != null) {
            instruments[0].pause();
        }
        if(instruments[1] != null) {
            instruments[1].pause();
        }
        if(instruments[2] != null) {
            instruments[2].pause();
        }
        if(instruments[3] != null) {
            instruments[3].pause();
        }
        if(instruments[4] != null) {
            instruments[4].pause();
        }

    }
});
rewindButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        if(instruments[0] != null) {
            instruments[0].stopAndRewind();
        }
        if(instruments[1] != null) {
            instruments[1].stopAndRewind();
        }
        if(instruments[2] != null) {
            instruments[2].stopAndRewind();
        }
        if(instruments[3] != null) {
            instruments[3].stopAndRewind();
        }
        if(instruments[4] != null) {
            instruments[4].stopAndRewind();
```

As you can see, I have wrapped each method call in an if statement. This means that the code will not throw up an error for attempting to run a method from an instrument that does not yet exist. This helps make my code error free, and more usable.

Testing and Analysis:

Test number	Description of test	Test data	Expected outcome	Achieved?
1	Test adding notes to one single instrument, and playing in instrumentSequencer	Input notes into individual instrument sequencer and click play	Notes should play in the correct order, at the correct pitch	YES
2	Test adding different types of instruments, and running playback	Add 1 instrument to the sequence and play. Repeat for different instruments	The type of noise played should sound different	YES
3	Test adding chords into the sequencer	Add an instrument, add chords to the sequence, then run playback	The chord's notes should play simultaneously	YES
4	Test multisequence threads work properly	Add multiple instruments, add notes to all instruments, and run playback on mainGUI	Notes from all instruments should play simultaneously	YES
5	Test pause, and resume buttons for individual instrument	Add an instrument, add notes to the instrument, play, pause, resume, pause, resume	When pause is pressed, the sequence should pause, and when resume is pressed, the sequence should resume	YES
6	Test pause, and resume buttons for multiple instruments	Add multiple instruments, add notes to all instruments,	When pause is pressed, the sequence should pause, and when	YES

		play, pause, resume, pause, resume	resume is pressed, the sequence should resume	

Analysis:

All tests resulted in the expected outcome, which is good since it means that my program has a fully functional user interface and sequencer. I did test my program throughout development, to make sure that all parts that I added to the GUI were added correctly, and all interactable components were able to be interacted with as expected. This is good because it helped me to notice errors during coding, which saved me a significant amount of time at the end of this stage.

The fact that I did iterative testing during the stage, and after I had finished the stage is good since it means that the user will have the experience that I intended on my program. This means that my program is more user-friendly and accessible for what my users will need to do.

On the other hand, I should have spent more time on developing pseudocode for more classes. This would have allowed me to save time during the development of the actual code. In future stages, I will spend more time developing the pseudocode, which will make my life a lot easier.

I also noticed that my stage 1 was a lot more complex than future stages. In retrospect, I should have broken down my program into more equal stages, which would have allowed me to easier understand what each stage entails, and therefore make planning each stage more manageable.

Stage 2 – Effects system:

In this stage, I will be creating 2 GUIs, one for the reverb effect, and one for the chorus effect. I will also need to add 2 classes: reverb, and chorus. When I combine these GUIs and classes with the rest of my program, it will allow the user to add different effects into their sequence and therefore give more availability to the user to sequence different styles of music. This is necessary for my system, because without it, my program would be too simple and would not provide enough features for the user to manipulate.

By the end of this stage, the user should be able to click on the effectsComboBox, and then select which effect they would like to add: reverb or chorus. From this, it should take the user to the respective GUI form, where they are able to change the how the effects impact the music – in terms of length, strength, and the applied-to instrument. From there, they will be able to return back to the mainGUI, play the music, and hear the differences to their previous sequence.

Design:

Algorithmic design:

- For the pseudocode, there is not any for the new GUI forms that I will add, since these forms will simply be creating the attributes for the components, and then calling methods from different classes inside listeners. This means that it is not necessary for me to create pseudocode for these parts, since there is going to be little and simple code that I can implement through IntelliJ's UI form manager.
- On the other hand, I will create pseudocode for the reverb and chorus class, since these will contain more of the logic for this section and therefore will require more planning to make the development of this stage easier. During programming of the reverb, I will scrap the reverb type I will first show the pseudocode for the reverb class and the chorus class:

```
Public class Reverb {

    Private String reverbType;

    Private int reverbStrength;

    Private int reverbLength;


    Private Instrument instrument;


    Public Reverb (String reverbType, reverbStrength, reverbLength) {

        This.reverbType = reverbType;

        This.reverbStrength = reverbStrength;

        This.reverbLength = reverbLength;

    }
    Public void applyReverb(Instrument instrument){

        Instrument.addReverb(this.reverbType, this.reverbStrength, this.revebLength);

    //Another method within the instrument class

    }

    Public void setReverbStrength(int reverbStrength){

        This.reverbStrength = reverbStrength;

    }

    Public void setReverbType(String reverbType){

        This.reverbType = reverbType;

    }

    Public void setReverbLength(int reverbLength){

        This.reverbLength = reverbLength;

    }

    Public void setApplyingInstrument(Instrument instrument){

        This.instrument = instrument;

    }

    Public int getReverbStrength(){

        Return this.reverbStrength;

    }

    Public String getReverbType(){

        Return this.reverbType;

    }

    Public int getReverbLength(){

        Return this.reverbLength;

    }
```

```
}


Public class Chorus{

        Private int modStrength;

        Private int modDifference;


        Public Chorus(int modStrength, int modDifference){

                This.modStrength = modStrength;

                This.modDifference = modDifference;

        }


        Public void setModStrength(int modStrangth){

                This.modStrength = modStrength;

        }

        Public void setModDifference(int modDifference){

                This.modDifference = modDifference;

        }

        Public setApplyingInstrument(Instrument instrument){

                This.instrument = instrument;

        }

        Public int getModDifference(){

                Return this.modDifference;

        }

        Public int getModStrength(){

                Return this.modStrength;

        }

        Public void applyChorus(Instrument instrument){

                Instrument.addChorus(this.modStrength, this.modDifference);
        }

}
```

- Next, I will show you a new method of the Instrument class that I will be adding. This method will contain all the code for applying the reverb effect to the sequence and is therefore necessary for this stage to go well. There will also be another new method in this class for adding the chorus effect as well. Here is the new code that will be added to instrument class:

```
Public class Instrument{

        Private Boolean reverbAdded = false;

        Private String reverbType;

        Private int reverbStrength;

        Private int reverbLength;


        Private Boolean chorusAdded = false;

        Private int modStrength;

        Private int modDifference;


        //ALL OTHER CODE PRE-EXISTING IN THIS CLASS//




        Public void addReverb(String reverbType, int reverbStrength, int
reverbLength){

                This.reverbAdded = true;

                This.reverbType = reverbType;

                This.reverbLength = reverbLength;

                This.reverbStrength = reverbStrength;

        }
```

```
    Public Boolean getReverbAdded(){
        Return reverbAdded;
    }
    Public void playNoteReverb(Reverb reverb, int note, int vol, int duration){
        int delay = reverb.getReverbLength();
        int strength = reverb.getReverbStrength() / 100;
        new Thread(){
            sleep(delay);
            noteOne(note, strength);
            sleep(delay);
            noteOff(note)
        }
        Thread.start();
    }

    Public void addChorus(modStrength, modDifference)(
        This.modStrength = modStrength;
        This.modDifference = modDifference;
        This.reverbAdded = true;
    }

    Public void playNoteChorus(Chorus chorus, int note, int duration, int vol){
        noteOn(note, vol);
        if(chorus != null){
            int modulationVolume = vol * (chorus.getModStrength() / 100)
            noteOn(note + chorus.getModDifference(),
modulationVolume);
```

```
                              noteOn(note – chorus.getModDifference(),
modulationVolume);
                    }
                    Sleep(duration);
                    If(chorus != null){
                              noteOff(note + chorus.getModDifference());
                              noteOff(note – chorus.getModDifference());
                    }
          }
}
```

Data:

Attributes:

Name	Type	Description	Held in class:	Local/Global
reverbType	String	Holds the type of reverb that is being applied	Reverb & Instrument	Global
reverbLength	Int	Holds the length that the reverb will last for	Reverb & Instrument	Global
reverbStrength	int	Holds the volume% of the original volume that reverb notes will be played at	Reverb & Instrument	global
modStrength	Int	Holds the volume% of	Chorus & Instrument	Global

		the original volume that chorus notes will be played at		
modDifference	Int	Holds the difference in semitones between the modulated notes and the original note	Chorus & Instrument	Global
reverbAdded	Boolean	Holds whether or not reverb is added to an instrument	Instrument	Global
chorusAdded	Boolean	Holds whether or not chorus is added to an instrument	Instrument	Global
Delay	Int	Holds the delay between regular notes and the reverberation	Instrument	Local to playNoteWithReverb method
Strength	Int	Holds the volume of reverbated notes	Instrument	Local to playNoteWithReverb method

modulationVolume	Int	Holds the volume of the modulated chorus notes	Instrument	Local to playNoteWithChorus method

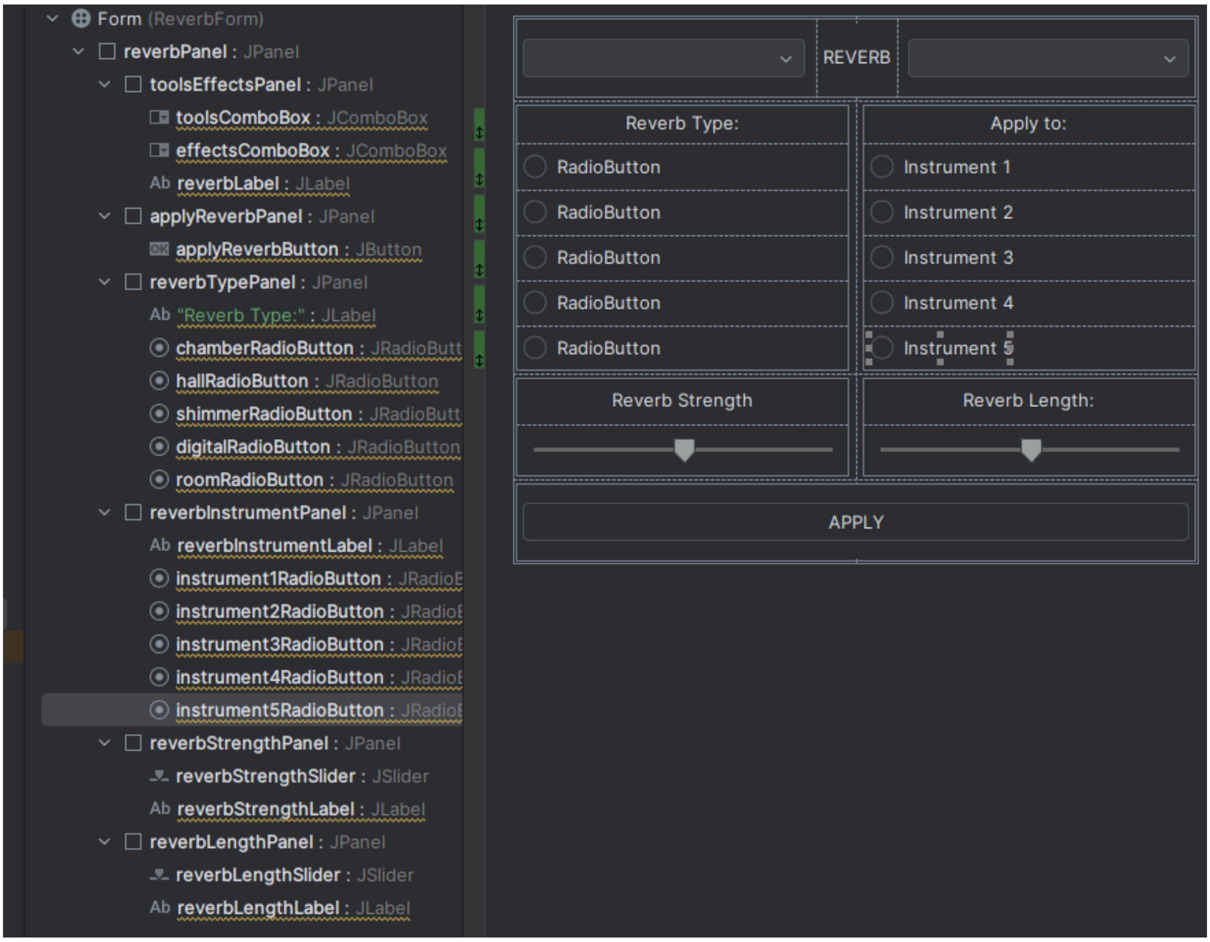
Methods:

Name	Return type	Description	Held in class:
applyReverb	Void	Adds reverb to an instrument	Reverb
setReverbLength	Void	Setter for reverbLength attribute	Reverb
setReverbStrength	Void	Setter for reverbStrength attribute	Reverb
setApplyingInstrument	Void	Setter for the instrument that reverb will be added to	Reverb
setReverbType	Void	Setter for the reverbType attribute	Reverb
getReverbStrength	Int	Getter for reverbStrength attribute	Reverb
getReverbType	String	Getter for reverbType attribute	Reverb
getReverbLength	Int	Getter for reverbLength attribute	Reverb
applyChorus	Void	Adds chorus to an instrument	Chorus
setModDifference	Void	Setter for modDifference attribute	Chorus

setModStrength	Void	Setter for modStrength attribute	Chorus
setApplyingInstrument	Void	Setter for the instrument that chorus will be applied to	Chorus
getModStrength	Int	Getter for modStrength attribute	Chorus
getModDifference	Int	Getter for modDifference attribute	Chorus
addReverb	Void	Adds reverb to an instrument and declares reverb related attributes in instrument class	Instrument
getReverbAdded	Boolean	Getter for reverbAdded attribute	Instrument
playNoteReverb	Void	Plays a note with reverb	Instrument
addChorus	Void	Adds reverb to an instrument and declares chorus related attributes in the instrument class	Instrument
getChorusAdded	Boolean	Getter for chorusAdded attribute	Instrument
playNoteChorus	void	Plays a note with the chorus effect	Instrument

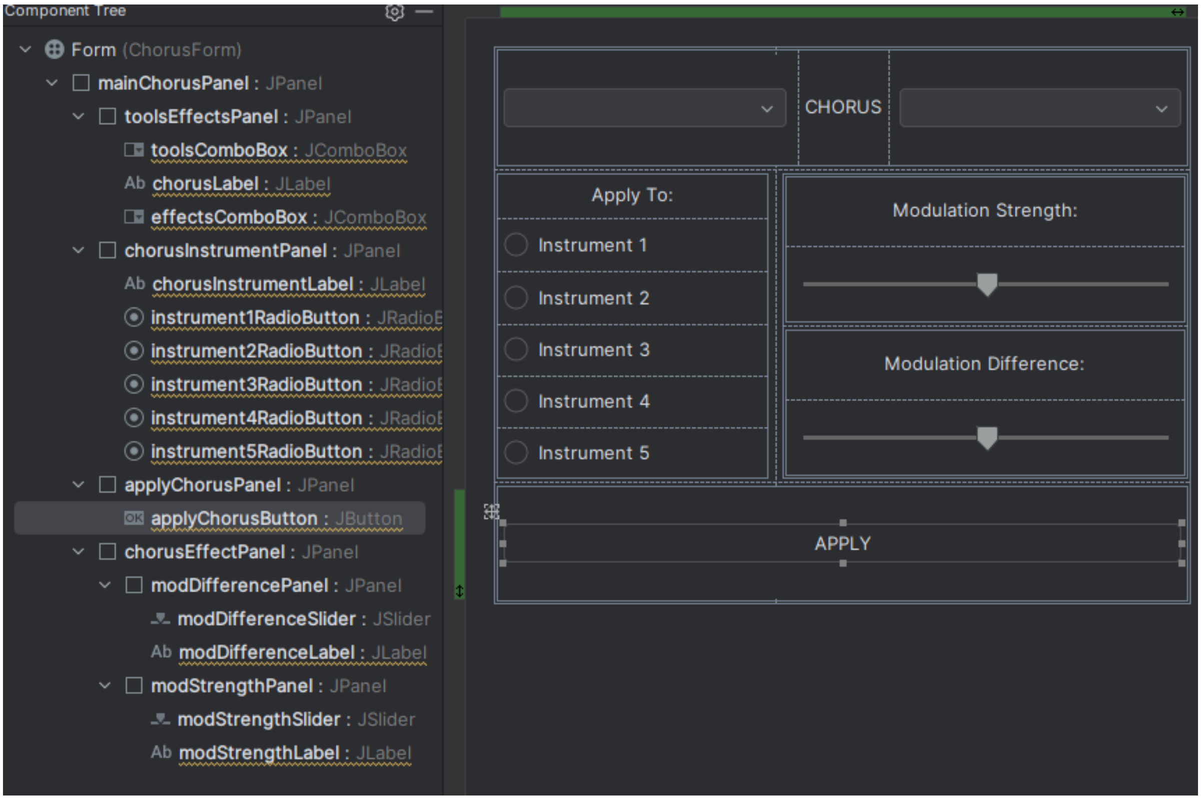
Development:

- First, I created all the classes and GUI forms that I will need to code this stage. This includes a GUI form for the chorus effect, a GUI form for the reverb effect, a reverb class and a chorus class. To do this, I looked back at my GUI designs and simply used that as a guide to where to put each component on the GUI. I used various components, such as JComboBox, JLabel, JSlider, JRadioButton, JButton and JPanel. I gave all components suitable names so that I can easily find out what each does later in coding. When I combined these components, this was the GUI for the reverbForm that I created:



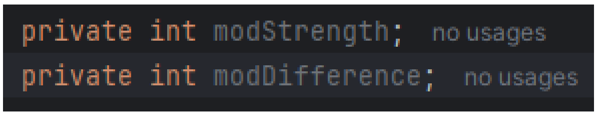
```
Form (ReverbForm)
  reverbPanel : JPanel
    toolsEffectsPanel : JPanel
      toolsComboBox : JComboBox
      effectsComboBox : JComboBox
      reverbLabel : JLabel
    applyReverbPanel : JPanel
      applyReverbButton : JButton
    reverbTypePanel : JPanel
      "Reverb Type:" : JLabel
      chamberRadioButton : JRadioButt
      hallRadioButton : JRadioButton
      shimmerRadioButton : JRadioButt
      digitalRadioButton : JRadioButton
      roomRadioButton : JRadioButton
    reverbInstrumentPanel : JPanel
      reverbInstrumentLabel : JLabel
      instrument1RadioButton : JRadioB
      instrument2RadioButton : JRadioB
      instrument3RadioButton : JRadioB
      instrument4RadioButton : JRadioB
      instrument5RadioButton : JRadioB
    reverbStrengthPanel : JPanel
      reverbStrengthSlider : JSlider
      reverbStrengthLabel : JLabel
    reverbLengthPanel : JPanel
      reverbLengthSlider : JSlider
      reverbLengthLabel : JLabel
```

Next, I started creating the GUI form for the Chorus class. I again referred to the GUI designs that I made prior to development. These designs were used to help me know where I should put each component, and the functionality that it should have (this will not yet be implemented however). I added various components, such as JComboBox, JLabel, JSlider, JRadioButton, JButton and JPanel. I gave all these components suitable names to help me understand their purpose during future coding. Here is the final GUI for the chorusForm:

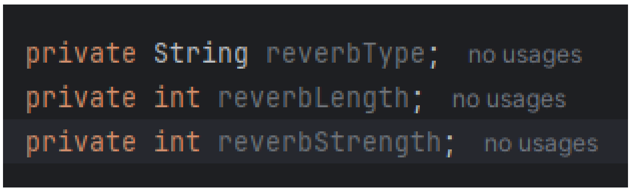


```
Component Tree
Form (ChorusForm)
  mainChorusPanel : JPanel
    toolsEffectsPanel : JPanel
      toolsComboBox : JComboBox
      chorusLabel : JLabel
      effectsComboBox : JComboBox
    chorusInstrumentPanel : JPanel
      chorusInstrumentLabel : JLabel
      instrument1RadioButton : JRadioB
      instrument2RadioButton : JRadioB
      instrument3RadioButton : JRadioB
      instrument4RadioButton : JRadioB
      instrument5RadioButton : JRadioB
    applyChorusPanel : JPanel
      applyChorusButton : JButton
    chorusEffectPanel : JPanel
      modDifferencePanel : JPanel
        modDifferenceSlider : JSlider
        modDifferenceLabel : JLabel
      modStrengthPanel : JPanel
        modStrengthSlider : JSlider
        modStrengthLabel : JLabel
```

I previewed both the chorusForm and the reverbForm, and they looked as expected and intended so this step was complete. After this preview had been done, I started by adding all the attributes for the reverb and chorus classes. To help me with this, I referred to my data dictionary for this stage. This allowed me to easier complete this section since all the names and data types of them were held in that table. Here are all the attributes declaration:

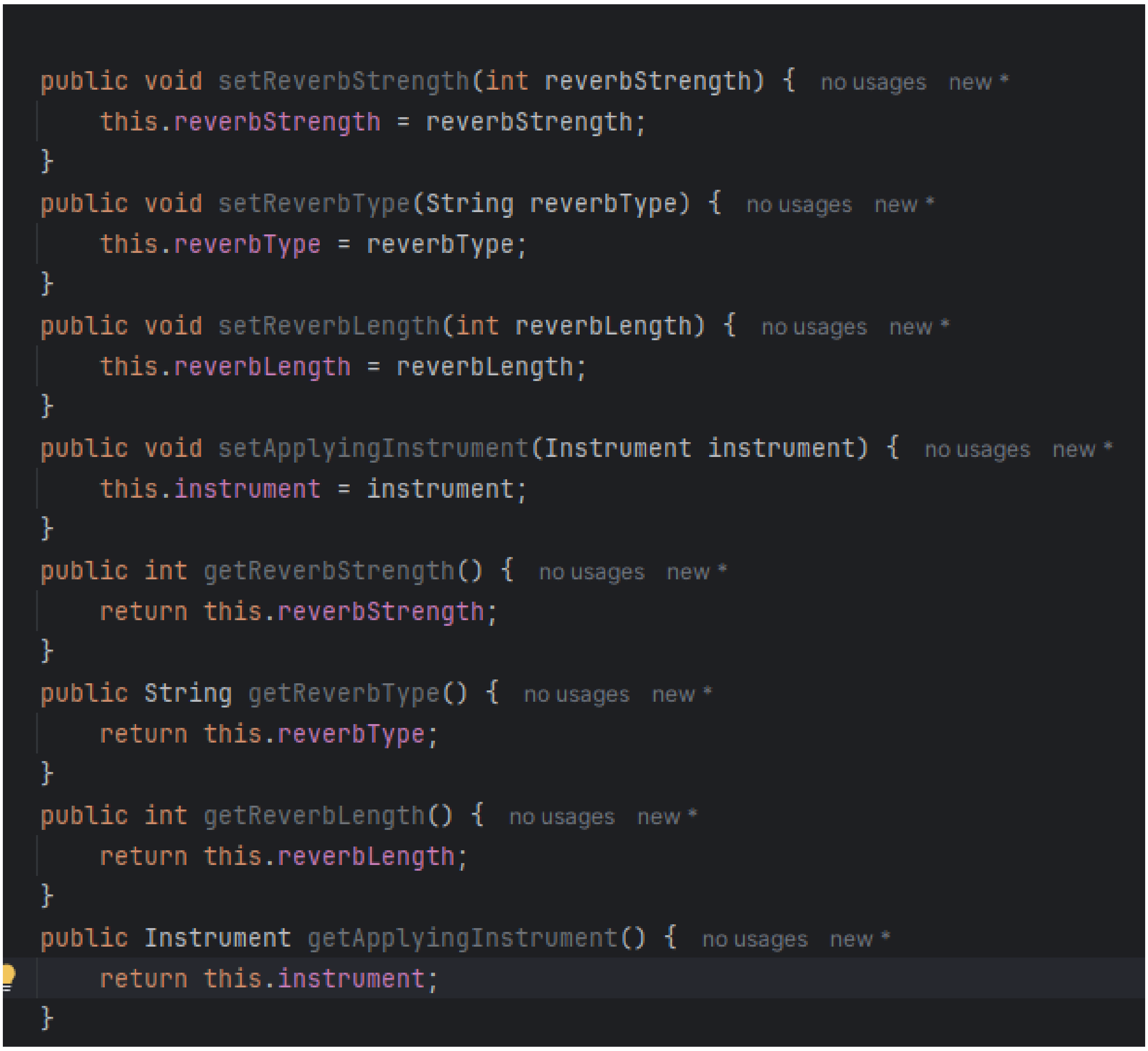


```java
private int modStrength;  no usages
private int modDifference;  no usages
```



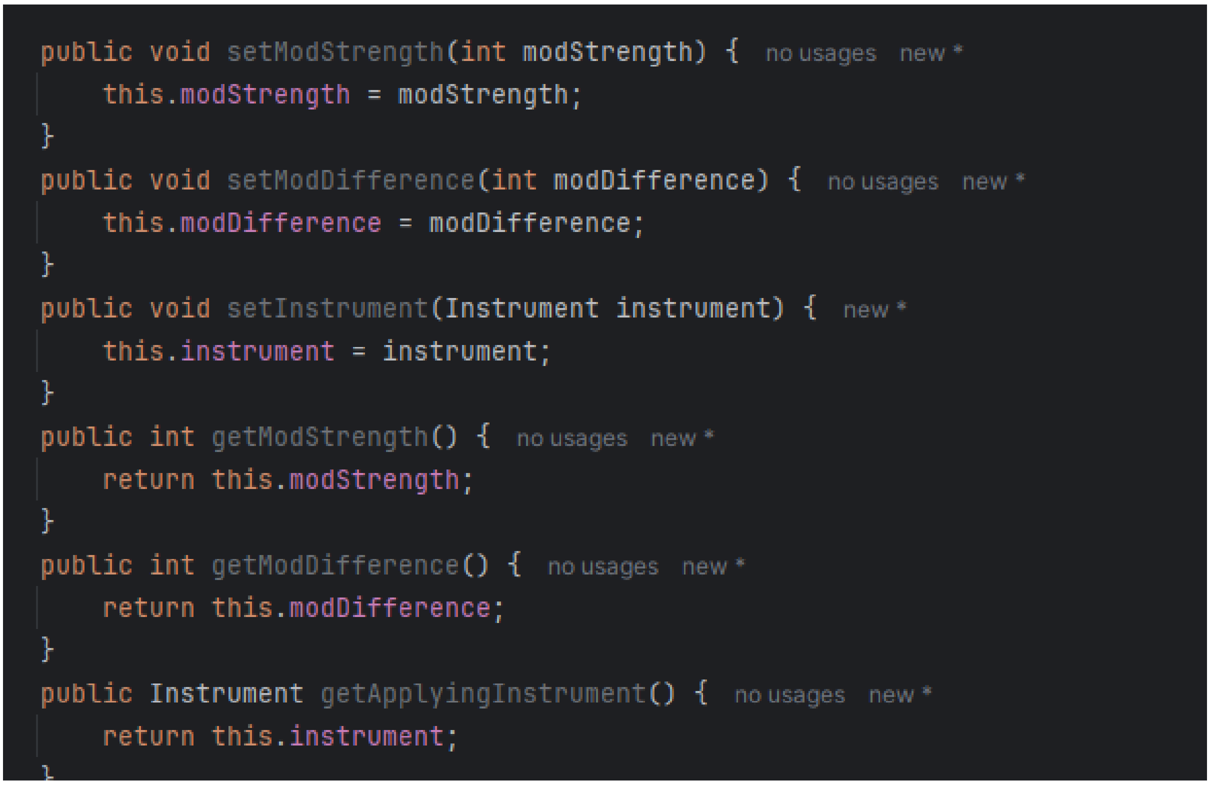
```java
private String reverbType;  no usages
private int reverbLength;  no usages
private int reverbStrength;  no usages
```

The next step was to create all the getters and setters for these attributes in the reverb class. These methods will allow me to access/change these attributes from anywhere in the program, so are particularly useful for playing an instrument with reverb. Here is the code for this section:



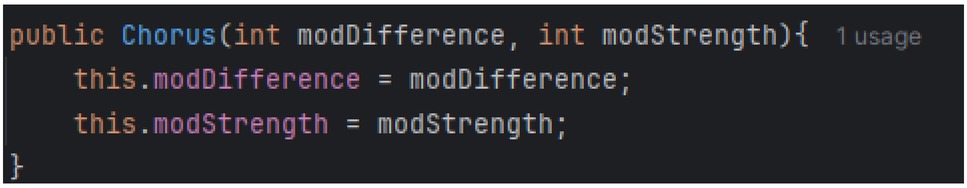
```
public void setReverbStrength(int reverbStrength) {  no usages  new *
    this.reverbStrength = reverbStrength;
}
public void setReverbType(String reverbType) {  no usages  new *
    this.reverbType = reverbType;
}
public void setReverbLength(int reverbLength) {  no usages  new *
    this.reverbLength = reverbLength;
}
public void setApplyingInstrument(Instrument instrument) {  no usages  new *
    this.instrument = instrument;
}
public int getReverbStrength() {  no usages  new *
    return this.reverbStrength;
}
public String getReverbType() {  no usages  new *
    return this.reverbType;
}
public int getReverbLength() {  no usages  new *
    return this.reverbLength;
}
public Instrument getApplyingInstrument() {  no usages  new *
    return this.instrument;
}
```

I am not yet able to test this code, since there is no functionality behind the reverb being applied or played yet. However, I am confident that these will work as they are relatively simple methods. I did this step for the chorus class too, since getting access to/changing these methods will be necessary for applying chorus and playing a note with chorus in future code. Here are all the getters and setters for the chorus class:

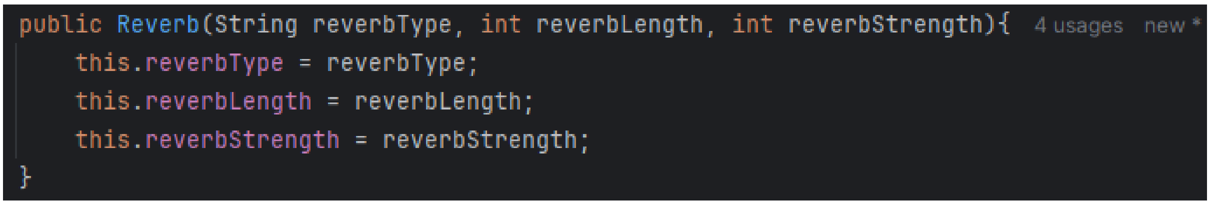


```
public void setModStrength(int modStrength) {  no usages  new *
    this.modStrength = modStrength;
}
public void setModDifference(int modDifference) {  no usages  new *
    this.modDifference = modDifference;
}
public void setInstrument(Instrument instrument) {  new *
    this.instrument = instrument;
}
public int getModStrength() {  no usages  new *
    return this.modStrength;
}
public int getModDifference() {  no usages  new *
    return this.modDifference;
}
public Instrument getApplyingInstrument() {  no usages  new *
    return this.instrument;
}
```

After that, I created the constructor for the reverb and chorus class. This special method allows me to declare the attributes of an object when it is created, which will be useful for setting the values for the length, strength, difference, etc. of the effects. Here is the code for both the chorus and reverb constructors:

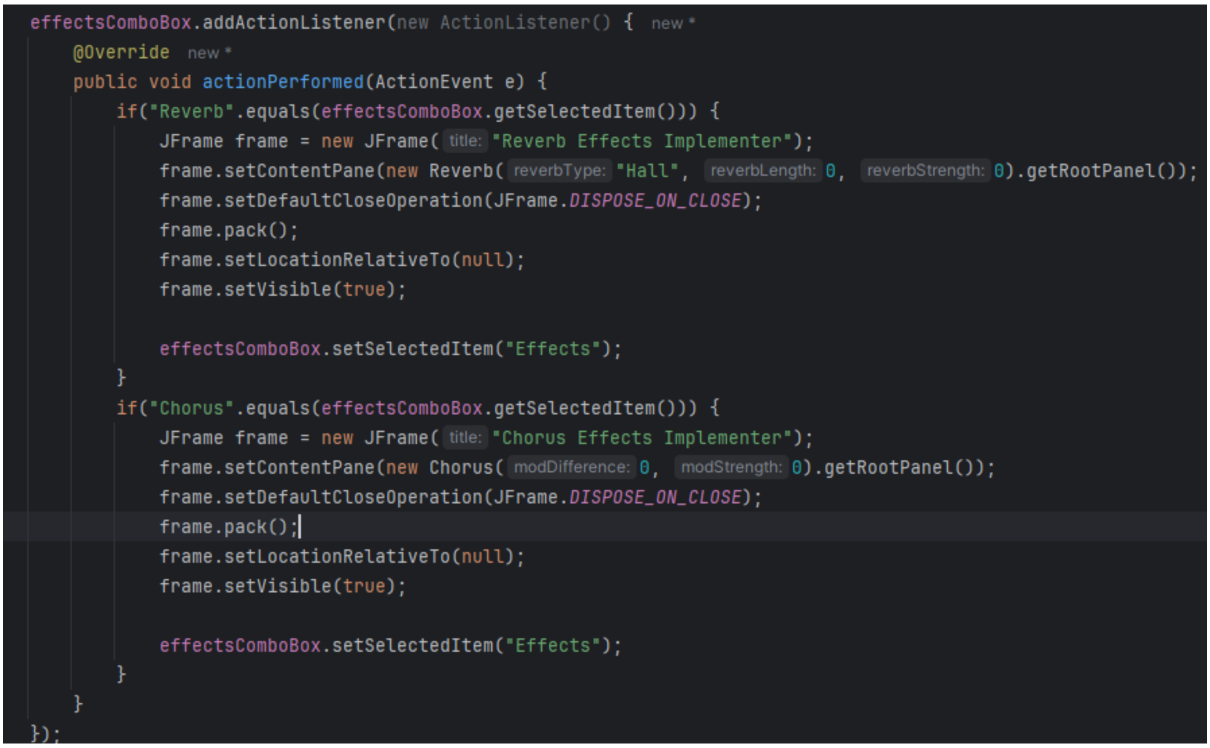


```
public Chorus(int modDifference, int modStrength){  1 usage
    this.modDifference = modDifference;
    this.modStrength = modStrength;
}
```



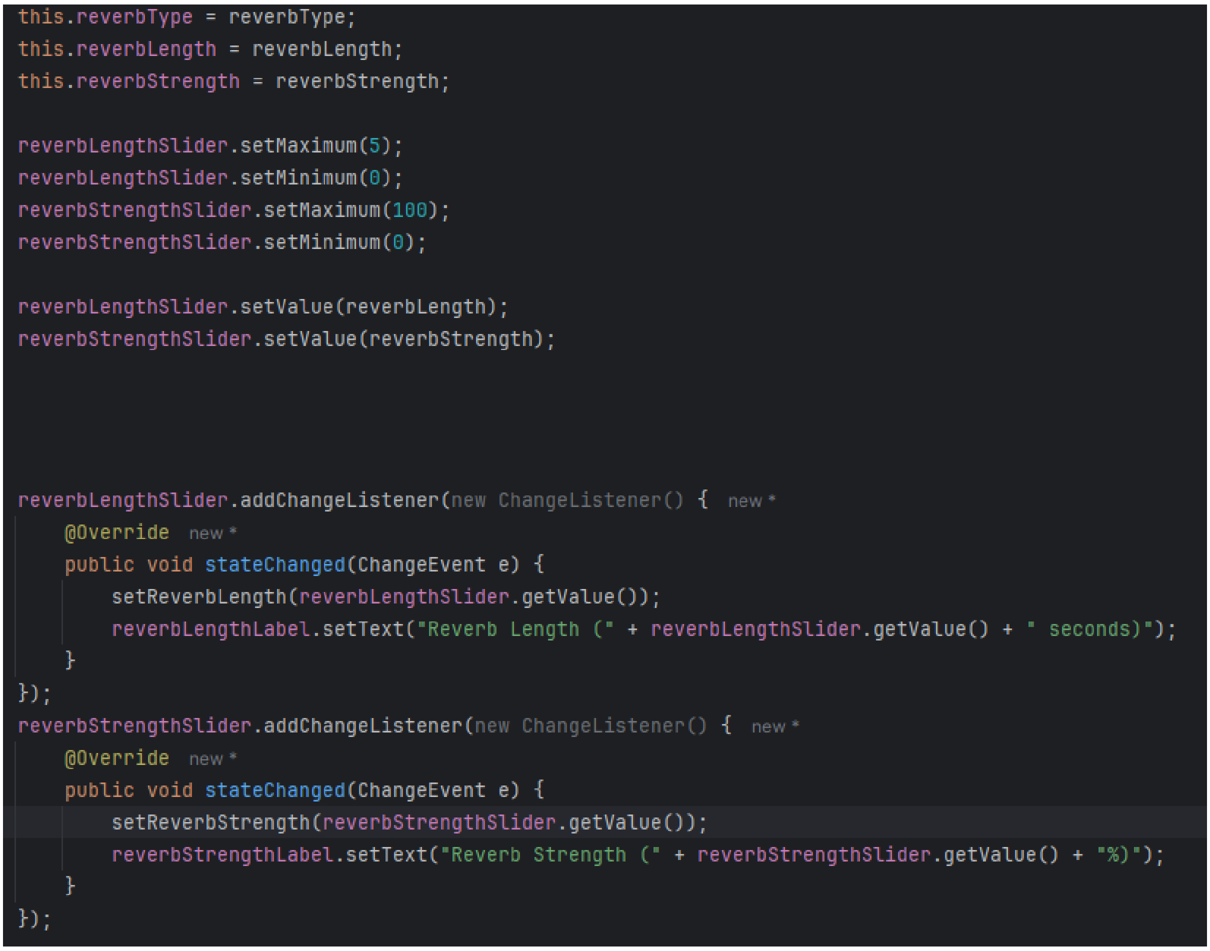
```
public Reverb(String reverbType, int reverbLength, int reverbStrength){  4 usages  new *
    this.reverbType = reverbType;
    this.reverbLength = reverbLength;
    this.reverbStrength = reverbStrength;
}
```

Once I had completed these special methods, I started linking the effectsComboBox in the mainGUI to their respective effects forms. This means that when the “Reverb” item is selected, the ReverbForm will open, and when the “Chorus” item is selected, the ChorusForm will open. After they are opened, the effectsComboBox in the mainGUI will return to the effects state. This means that when the mainGUI is opened again, the effectsComboBox will be able to be used again. This is the code for that section:



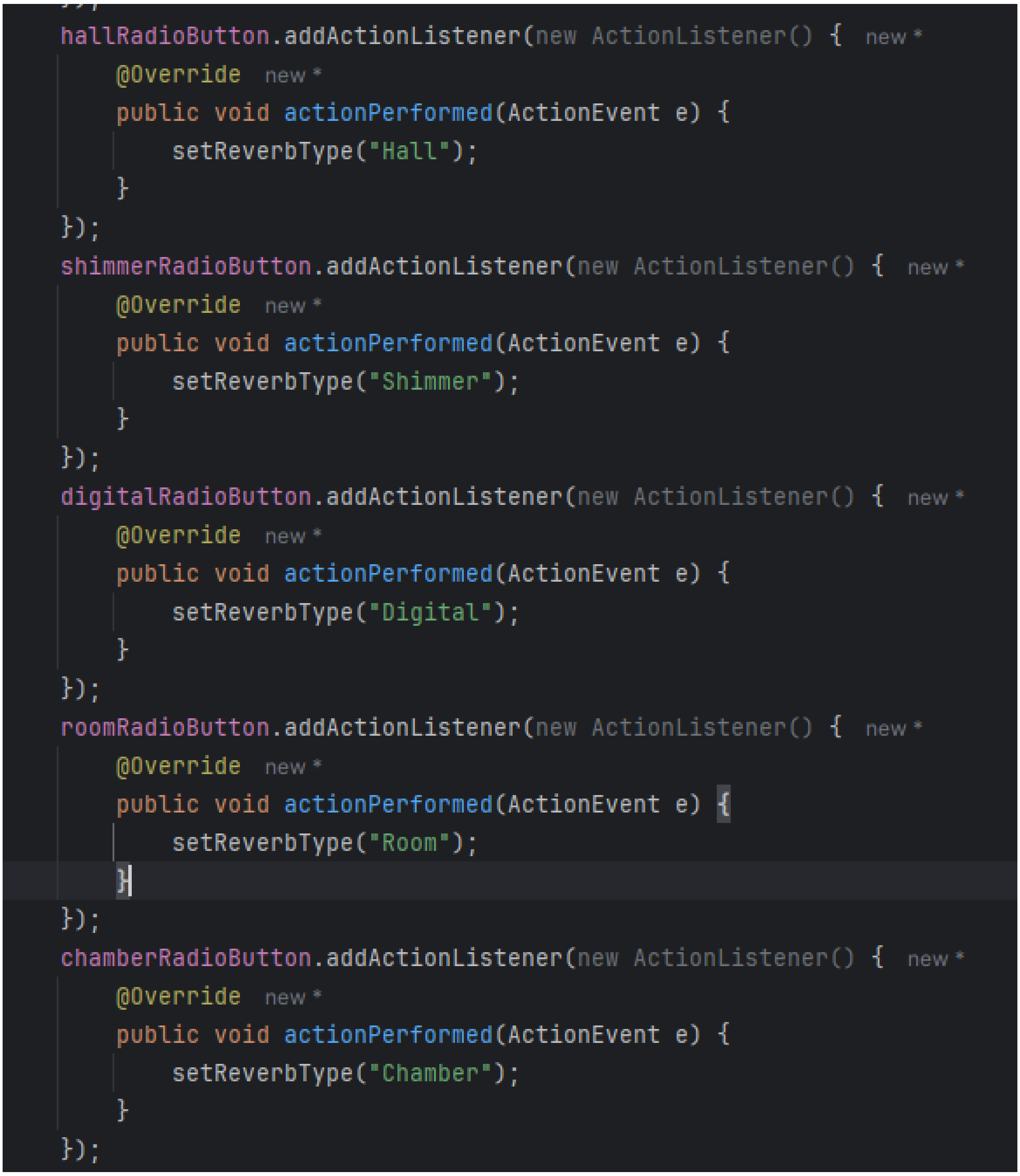
```
effectsComboBox.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        if("Reverb".equals(effectsComboBox.getSelectedItem())) {
            JFrame frame = new JFrame( title: "Reverb Effects Implementer");
            frame.setContentPane(new Reverb( reverbType: "Hall",  reverbLength: 0,  reverbStrength: 0).getRootPanel());
            frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);

            effectsComboBox.setSelectedItem("Effects");
        }
        if("Chorus".equals(effectsComboBox.getSelectedItem())) {
            JFrame frame = new JFrame( title: "Chorus Effects Implementer");
            frame.setContentPane(new Chorus( modDifference: 0,  modStrength: 0).getRootPanel());
            frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setVisible(true);

            effectsComboBox.setSelectedItem("Effects");
        }
    }
});
```

I ran the code and attempted to access the chorus and reverb forms via the effectsComboBox, and it worked so I was happy that this section was functional. Next, I had to create Change Listeners to change the different features of the effects. I did this for the reverb effect first to see how these features would function. I also set the text of the reverbLengthLabel and reverbStrengthLabel inside of these listeners. This allowed me to show the user what the current values of their reverbLength and reverbStrength are. This will give the user more availability to manipulate their program as they intend. Here is the code for this part:



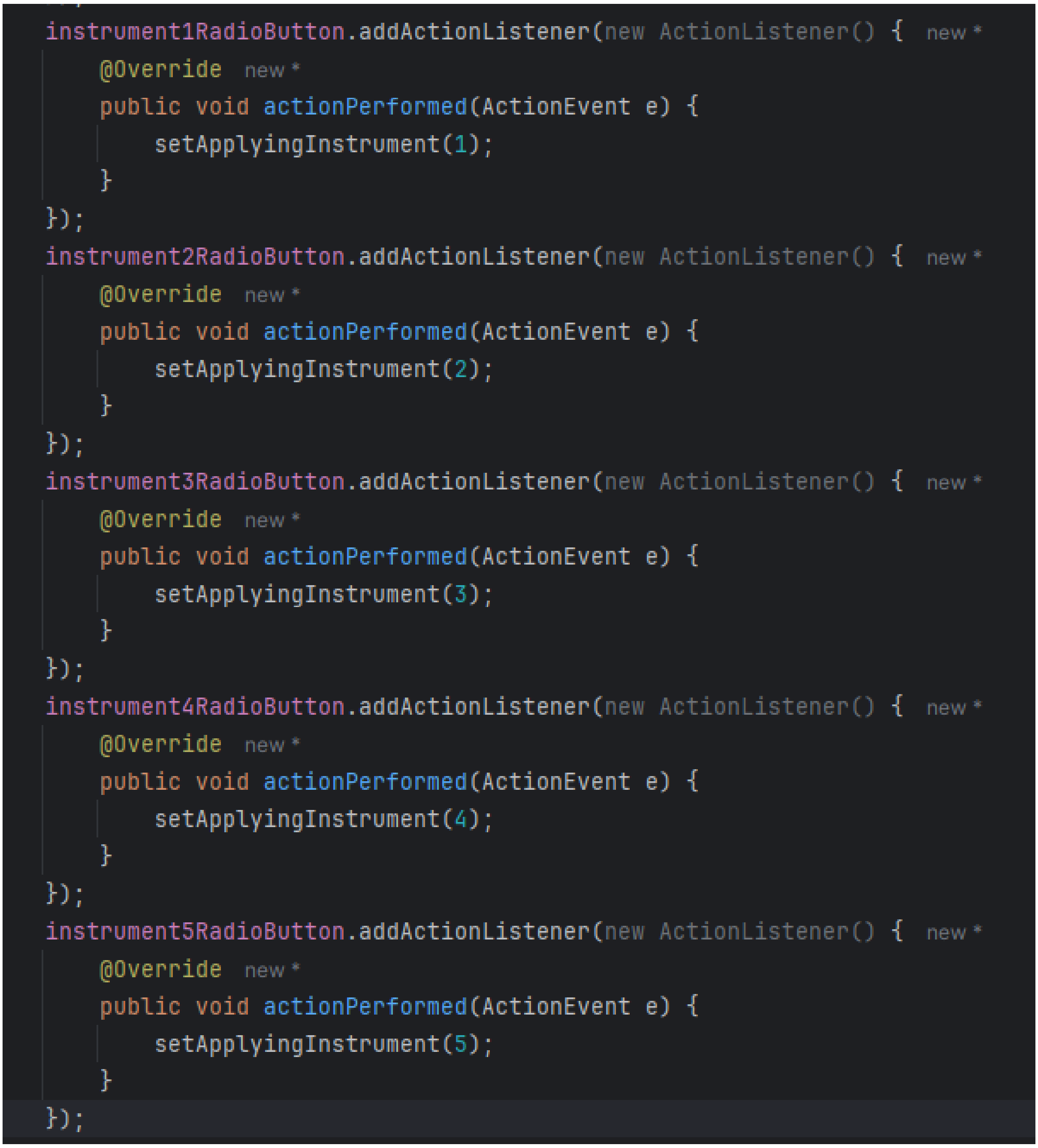
```
this.reverbType = reverbType;
this.reverbLength = reverbLength;
this.reverbStrength = reverbStrength;

reverbLengthSlider.setMaximum(5);
reverbLengthSlider.setMinimum(0);
reverbStrengthSlider.setMaximum(100);
reverbStrengthSlider.setMinimum(0);

reverbLengthSlider.setValue(reverbLength);
reverbStrengthSlider.setValue(reverbStrength);




reverbLengthSlider.addChangeListener(new ChangeListener() {  new *
    @Override  new *
    public void stateChanged(ChangeEvent e) {
        setReverbLength(reverbLengthSlider.getValue());
        reverbLengthLabel.setText("Reverb Length (" + reverbLengthSlider.getValue() + " seconds)");
    }
});
reverbStrengthSlider.addChangeListener(new ChangeListener() {  new *
    @Override  new *
    public void stateChanged(ChangeEvent e) {
        setReverbStrength(reverbStrengthSlider.getValue());
        reverbStrengthLabel.setText("Reverb Strength (" + reverbStrengthSlider.getValue() + "%)");
    }
});
```

I tested this within my reverbForm and it worked as expected (i.e the text changed for the reverbStrengthLabel and reverbLengthLabel to show what the values of the respective attributes had changed). After that, I had to create Action Listeners for all the JRadioButtons to set the values for other attributes within the reverb class (reverbType and instrument). This will mean that I have all attributes ready for the user to set, meaning they can add their choices of reverb to the sequence. This was relatively simple as it was simply grouping all the radio buttons together and then setting values for each of them within an action listener. Here is the code for the reverbTypeRadioButtonGroup:



```
hallRadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setReverbType("Hall");
    }
});
shimmerRadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setReverbType("Shimmer");
    }
});
digitalRadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setReverbType("Digital");
    }
});
roomRadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setReverbType("Room");
    }
});
chamberRadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setReverbType("Chamber");
    }
});
```

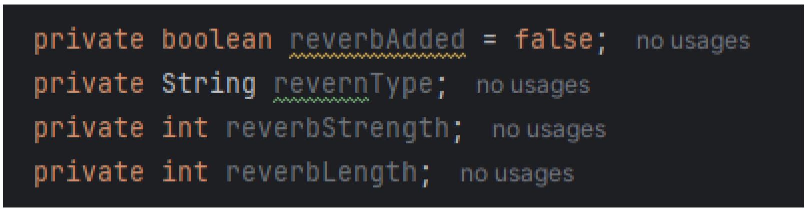
As you can see this is relatively simple code, as I only must call the setReverbType method to change the attribute reverbType with respect to what button has been selected. I used this same methodology to set the value of the applying instrument so that the user can change what instrument they would like to apply the reverb to, with the setting for the effect that they choose. Here is the code:



```
instrument1RadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setApplyingInstrument(1);
    }
});
instrument2RadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setApplyingInstrument(2);
    }
});
instrument3RadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setApplyingInstrument(3);
    }
});
instrument4RadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setApplyingInstrument(4);
    }
});
instrument5RadioButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        setApplyingInstrument(5);
    }
});
```

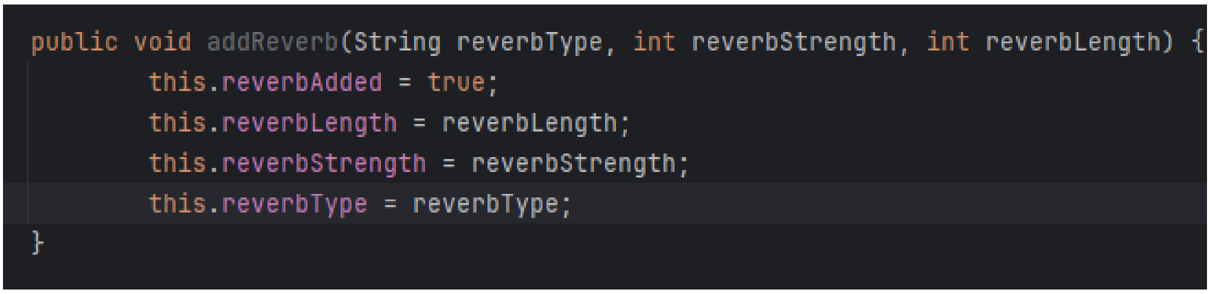
This code looks like the last block of code, by adding an action listener for each radio button, which sets the number of the applying instrument to a number relating to the index of the instrument.

The next segment that I programmed was declaring the variables for the reverb-related attributes within the instrument class. This stage will allow me to add more code to add reverb to an instrument, before playing it. I have already declared reverbAdded as false, since reverb is an optional effect, meaning that it will not be-added to a sequence. Here is the code for this part:



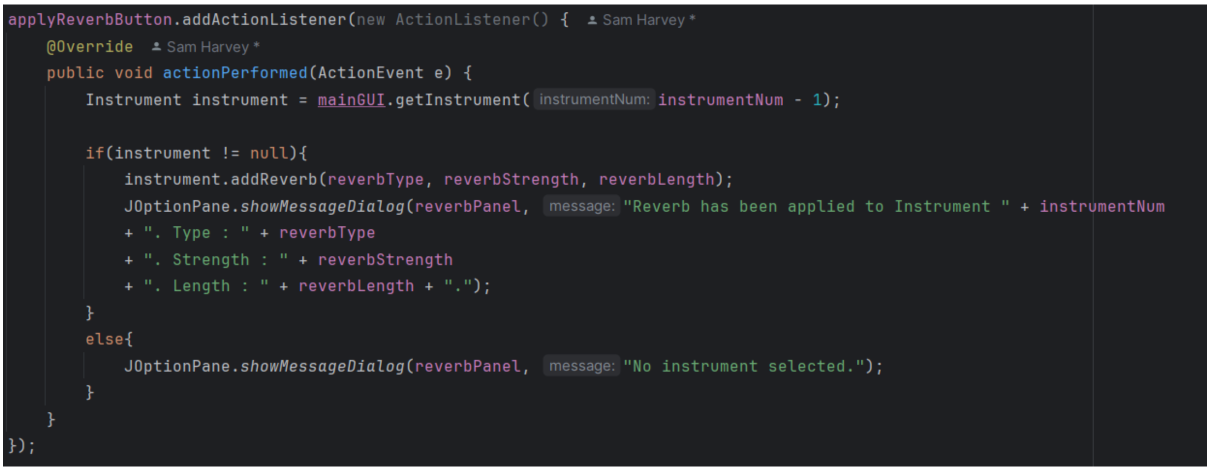
```
private boolean reverbAdded = false;  no usages
private String revernType;  no usages
private int reverbStrength;  no usages
private int reverbLength;  no usages
```

Next, I created the addReverb method within the instrument class. This method will allow me to create a reverb object and then relate it to an instrument. This was simply done by setting all the attributes that are related to reverb to those that are passed into the method. Here is the code for that stage:



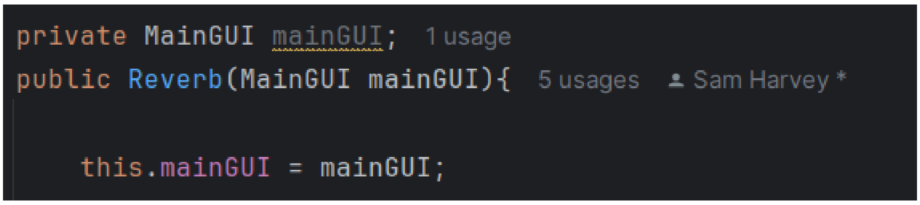
```
public void addReverb(String reverbType, int reverbStrength, int reverbLength) {
        this.reverbAdded = true;
        this.reverbLength = reverbLength;
        this.reverbStrength = reverbStrength;
        this.reverbType = reverbType;
}
```

The next step for me was to add code within an action Listener for the applyReverbButton, which allows me to run the addReverb method that is held within the instrument class. This means that variables in the Instrument class can be set to the correct values and reverbAdded can be set to true. This means that the user is unfunctionally able to add reverb into their sequence. In this block of code, I will also add a label which displays to the user that reverb has been added. This means that the user will not think that the form has been closed, but instead that reverb has been added. Since this message will be displayed on the JOptionPane, I had to learn how to use this feature, so I referred to geekstogeeks to help me with this (https://www.geeksforgeeks.org/java/java-joptionpane/). This makes my program more usable. Here is the code for that section:



```
applyReverbButton.addActionListener(new ActionListener() {  Sam Harvey *
    @Override  Sam Harvey *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument( instrumentNum: instrumentNum - 1);

        if(instrument != null){
            instrument.addReverb(reverbType, reverbStrength, reverbLength);
            JOptionPane.showMessageDialog(reverbPanel,  message: "Reverb has been applied to Instrument " + instrumentNum
            + ". Type : " + reverbType
            + ". Strength : " + reverbStrength
            + ". Length : " + reverbLength + ".");
        }
        else{
            JOptionPane.showMessageDialog(reverbPanel,  message: "No instrument selected.");
        }
    }
});
```

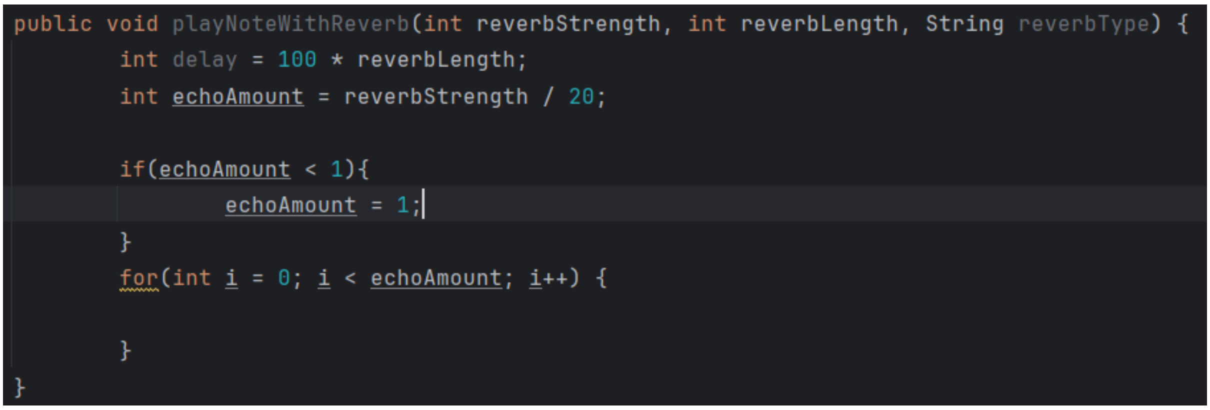
I tested this and it did not work. I then realised that I had not declared the mainGUI so I was not able to run the addReverb method. I went back into my constructor for the reverb class and changed. I did this so that I could run the addReverb method and therefore apply reverb to an instrument. This makes my program more usable. These are the changes that I made:



```
private MainGUI mainGUI;  1 usage
public Reverb(MainGUI mainGUI){  5 usages  Sam Harvey *

    this.mainGUI = mainGUI;
```

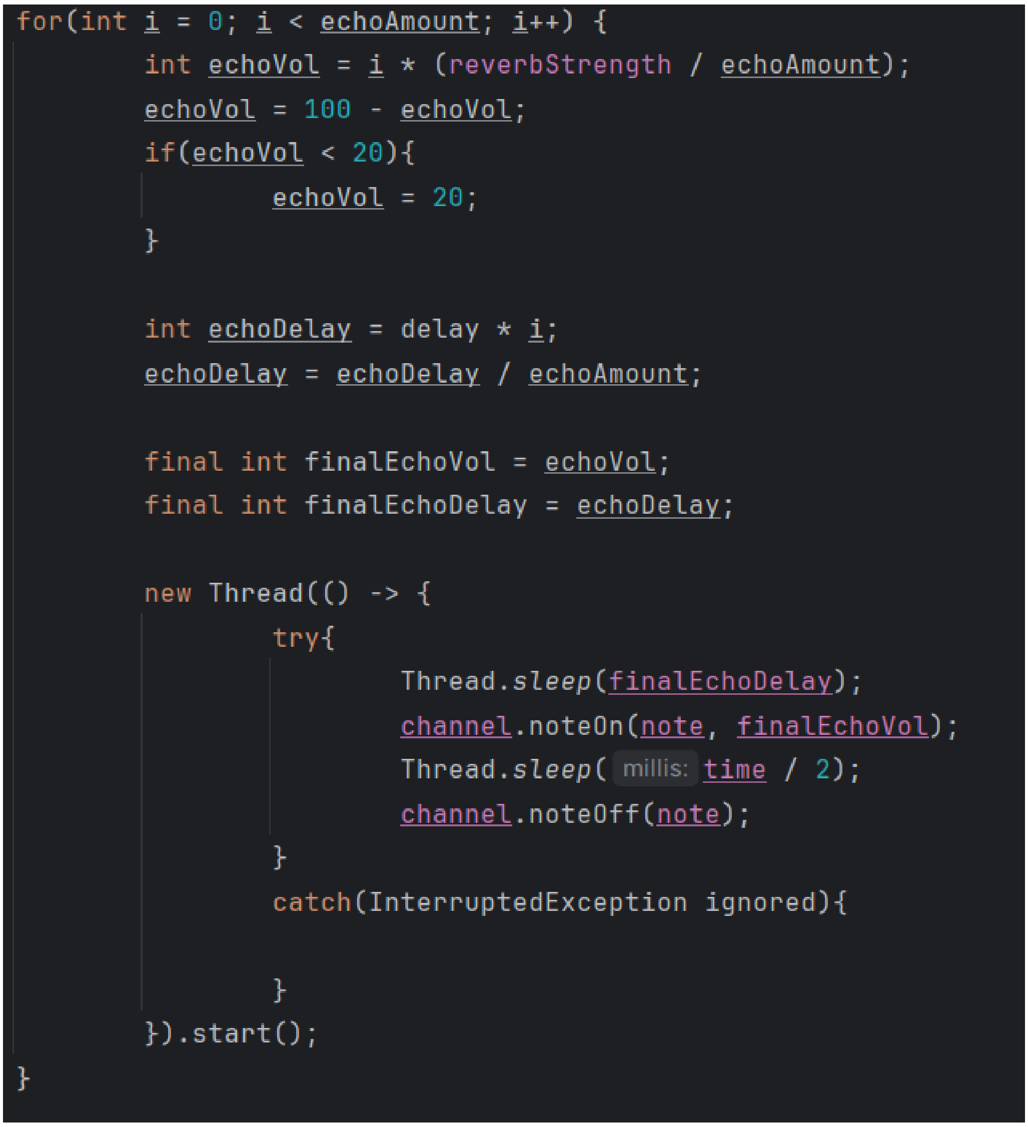
I tested this again and it worked (i.e. the JOptionPane showed up with a message saying the reverb had been added to the desired instrument with desired attributes). This implies that reverb has been added.

The next step was to add a method for playing a note with reverb. This means that I could check if reverb had been added and if it had, the correct method would be played to play that note with reverb and if it has not, the regular method to play a note without reverb will be added. This means that the reverb effect will have full functionality, meaning that the user has more ability to manipulate their music however they feel. Here is the start of the code for this section:



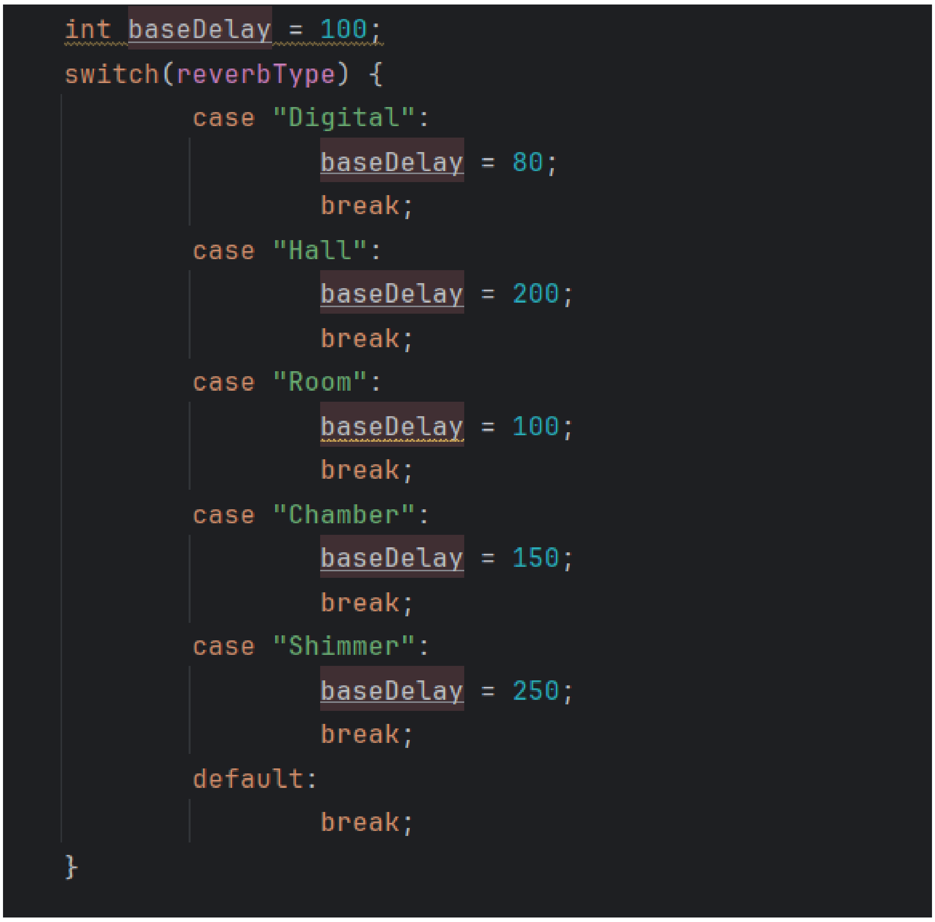
```
public void playNoteWithReverb(int reverbStrength, int reverbLength, String reverbType) {
        int delay = 100 * reverbLength;
        int echoAmount = reverbStrength / 20;

        if(echoAmount < 1){
                echoAmount = 1;
        }
        for(int i = 0; i < echoAmount; i++) {

        }
}
```

As you can see, I have added variables for the delay between echoes and the number of echoes that will be played. I have also added an if statement which states that the number of echoes is less than 1, echoAmount is set back to 1. This is don't to make sure that at least one echo is played even when the reverbStrength is not large enough to do that. The next step was to code in a system to play the echoes after each not. This was don't by using threads again, so I had to refer to geeksforgeeks (https://www.geeksforgeeks.org/java/multithreading-in-java/) to help me with the use of threads. The purpose of using threads in this section was to allow both the regular note and echoes to play at the same time via multithreading. Here was the code for this part:



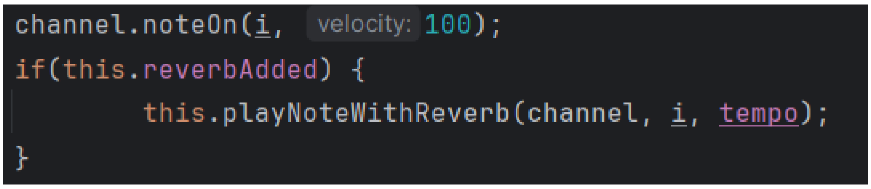
```
for(int i = 0; i < echoAmount; i++) {
        int echoVol = i * (reverbStrength / echoAmount);
        echoVol = 100 - echoVol;
        if(echoVol < 20){
                echoVol = 20;
        }

        int echoDelay = delay * i;
        echoDelay = echoDelay / echoAmount;

        final int finalEchoVol = echoVol;
        final int finalEchoDelay = echoDelay;

        new Thread(() -> {
                try{
                        Thread.sleep(finalEchoDelay);
                        channel.noteOn(note, finalEchoVol);
                        Thread.sleep( millis: time / 2);
                        channel.noteOff(note);
                }
                catch(InterruptedException ignored){

                }
        }).start();
}
```

As you can see, when a reverbed note is played, the echoVol is set to the 100 – (number of echo) * (strength of the reverb / the number of echoes). To make sure all echoes can be heard, I set this value to 20 after it has gotten below 20. The echoDelay (the amount of time in ms between each echo) is then set to the regular reverb delay * (the echo number) / (the number of total echoes). After all these variables are set to the correct value, a thread is started to play a note at decreasing volume for each value of i. This is how simple reverb works (i.e. playing a quieter not at intervals before the original note played). After I had finished coding that, I realised that I had not implemented any feature to involve the reverbType attribute. I coded in a way of deciding the baseDelay for each type of reverb, so that different types had different base effects that the reverb had. Here was the code for that part:



```
int baseDelay = 100;
switch(reverbType) {
        case "Digital":
                baseDelay = 80;
                break;
        case "Hall":
                baseDelay = 200;
                break;
        case "Room":
                baseDelay = 100;
                break;
        case "Chamber":
                baseDelay = 150;
                break;
        case "Shimmer":
                baseDelay = 250;
                break;
        default:
                break;
}
```

I then implemented this method into the play() method within the instrument class so that I could test it. This was done relatively simply, just by adding a way of checking if reverb is added to that instrument, and if it had you would run the method. This was done just after the note had been to reduce the delay between reverbed echoes and the original note played. Here was the code for that section:



```
channel.noteOn(i,  velocity: 100);
if(this.reverbAdded) {
        this.playNoteWithReverb(channel, i, tempo);
}
```

I then tested this implementation for different values of reverbLength, reverbStrength and reverbType and it worked as expected (i.e. the notes reverbed at different times/strengths/volumes depending on the values that I inputted after adding notes to an instrument.