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

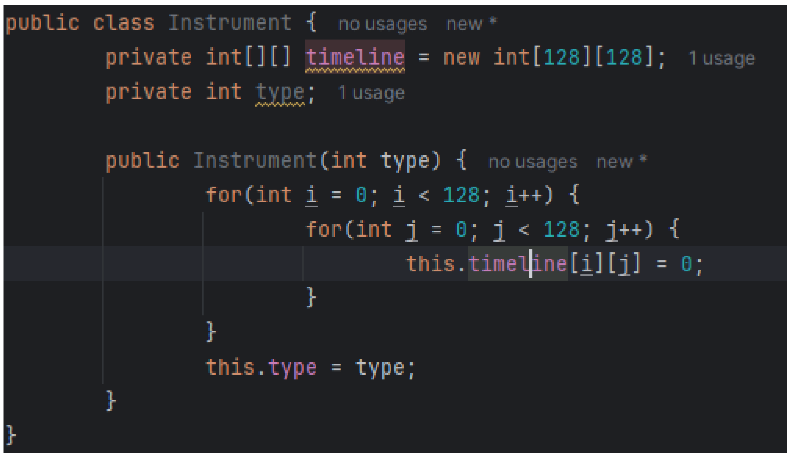
Methods:

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

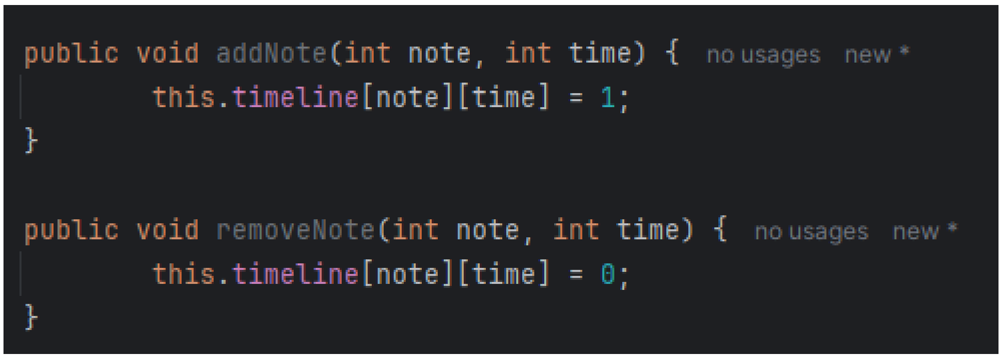
Development:

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:



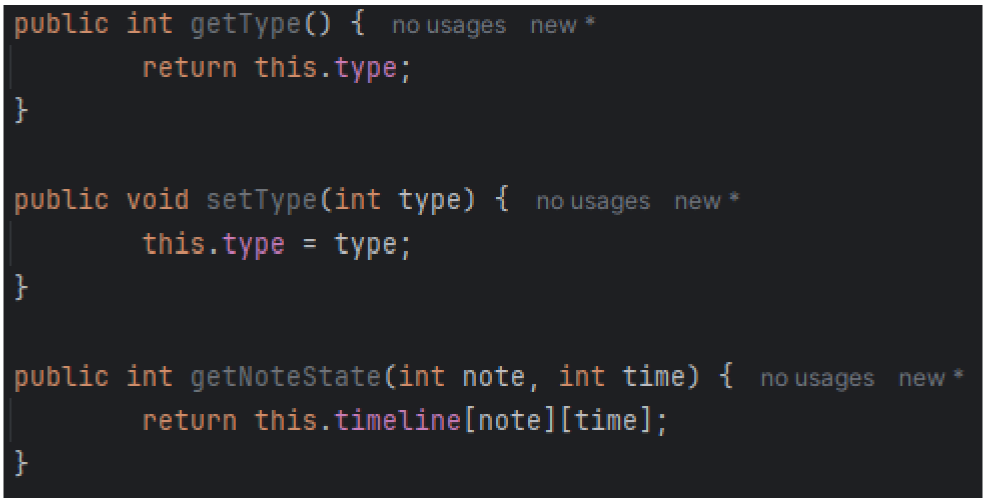
```
public class Instrument {  no usages  new *
        private int[][] timeline = new int[128][128];  1 usage
        private int type;  1 usage

        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:



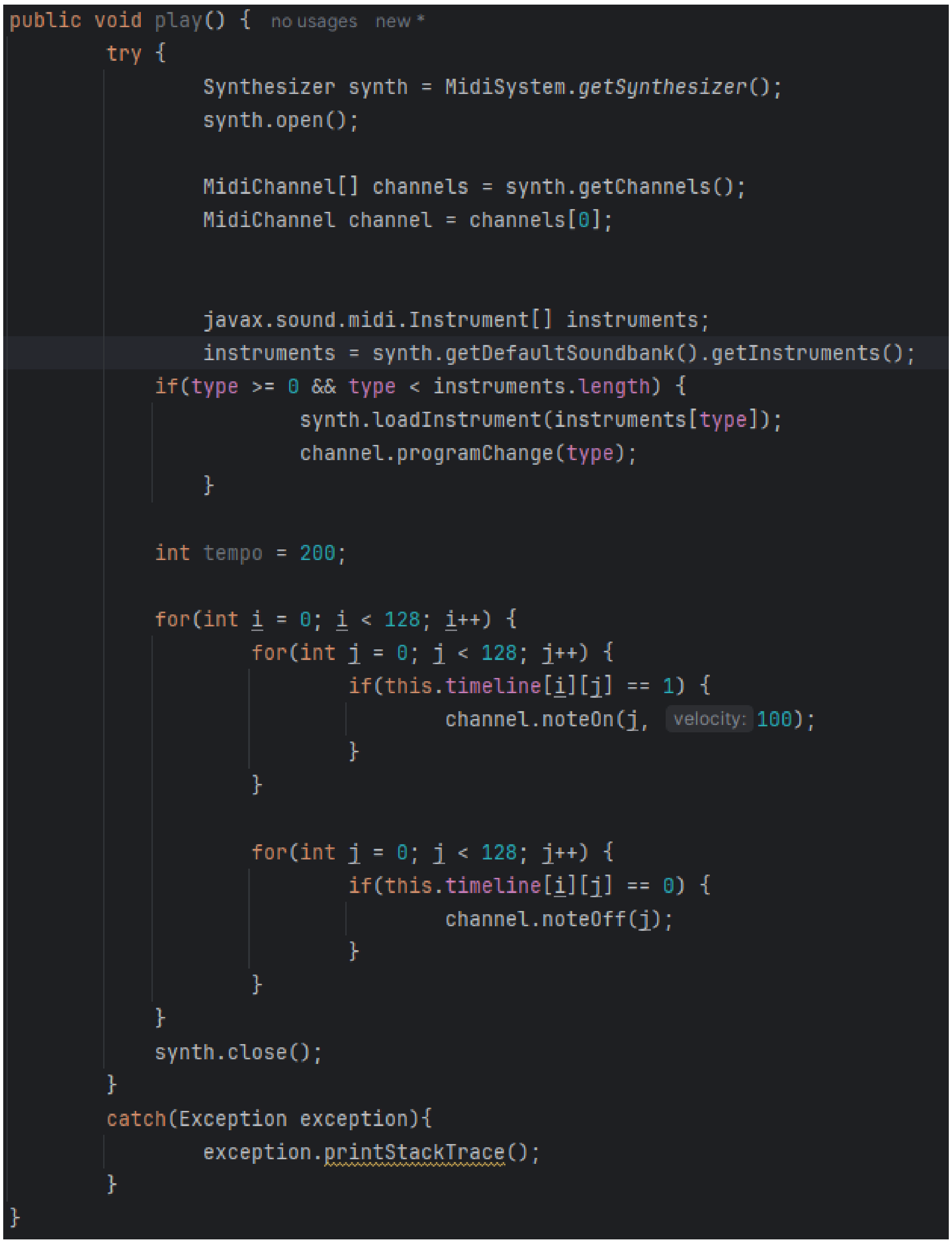
```
public void addNote(int note, int time) {  no usages  new *
        this.timeline[note][time] = 1;
}

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() {  no usages  new *
        return this.type;
}

public void setType(int type) {  no usages  new *
        this.type = type;
}

public int getNoteState(int note, int time) {  no usages  new *
        return this.timeline[note][time];
}
```

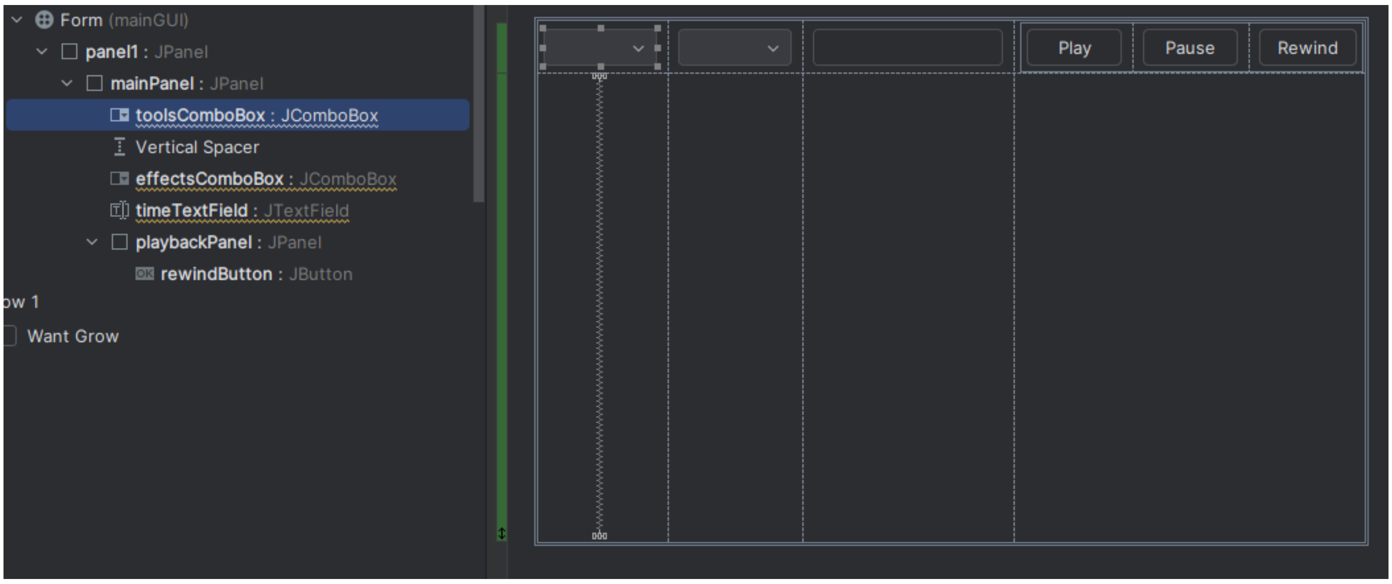
After that, I started creating the play method. For this I had to learn how to create and manipulate sound files, so I looked at REINTECH (https://reintech.io/blog/java-midi-programming-creating-manipulating-midi-data), and rememberjava (https://rememberjava.com/midi/2017/01/13/midi_basics.html) which 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:



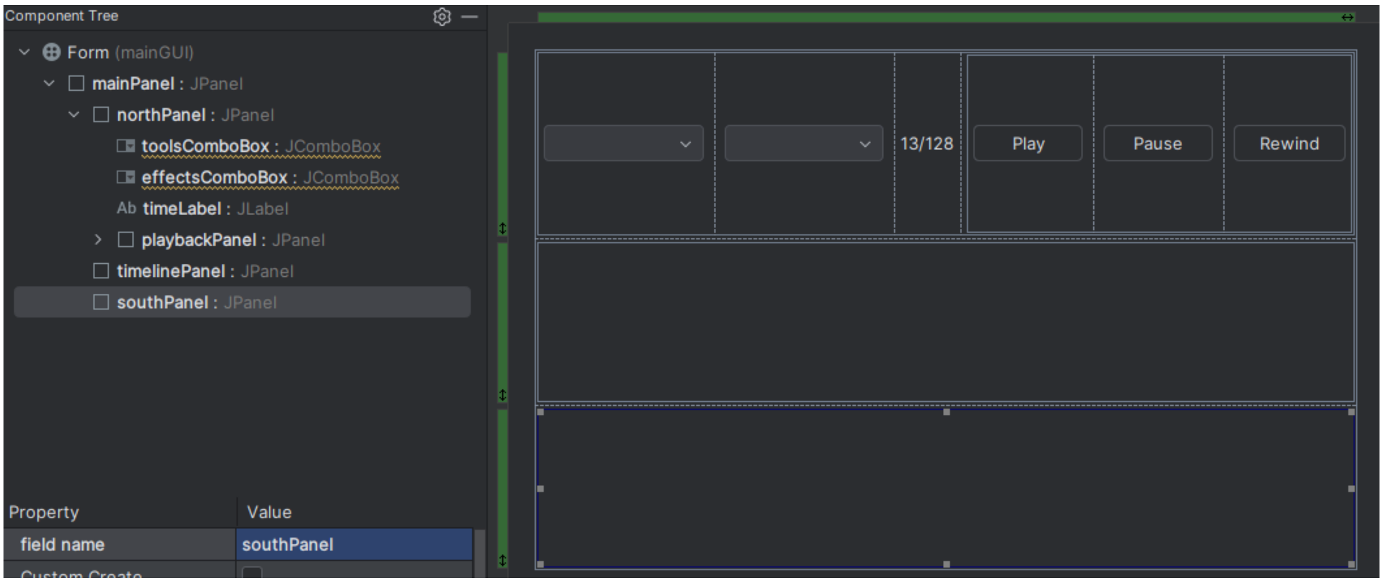
```
public void play() {  no usages  new *
        try {
            Synthesizer synth = MidiSystem.getSynthesizer();
            synth.open();

            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:

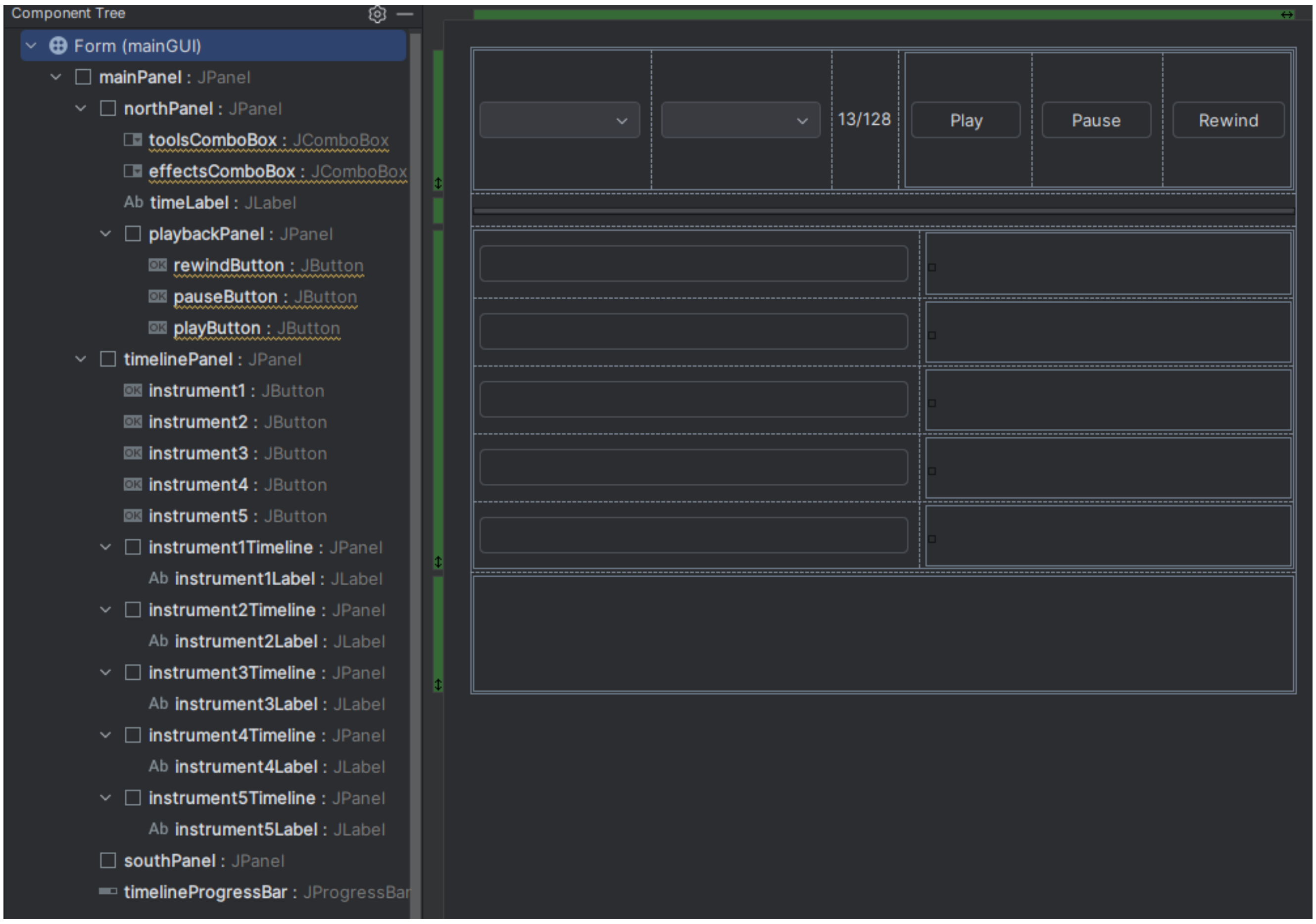

Form (mainGUI)
panel1 : JPanel
mainPanel : JPanel
toolsComboBox : JComboBox
Vertical Spacer
effectsComboBox : JComboBox
timeTextField : JTextField
playbackPanel : JPanel
rewindButton : JButton
ow 1
Want Grow

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:

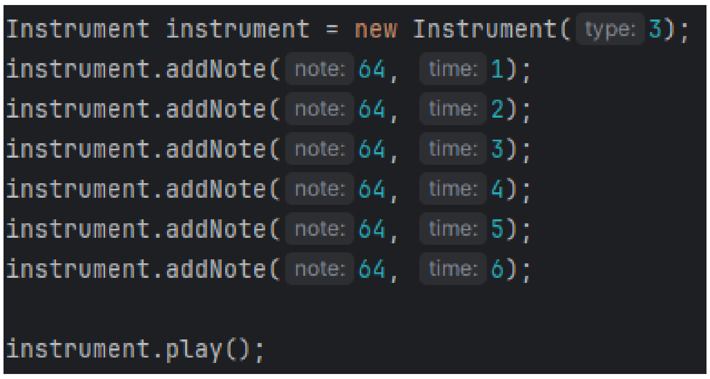

Component Tree
Form (mainGUI)
mainPanel : JPanel
northPanel : JPanel
toolsComboBox : JComboBox
effectsComboBox : JComboBox
timeLabel : JLabel
playbackPanel : JPanel
timelinePanel : JPanel
southPanel : JPanel

Property	Value
field name	southPanel

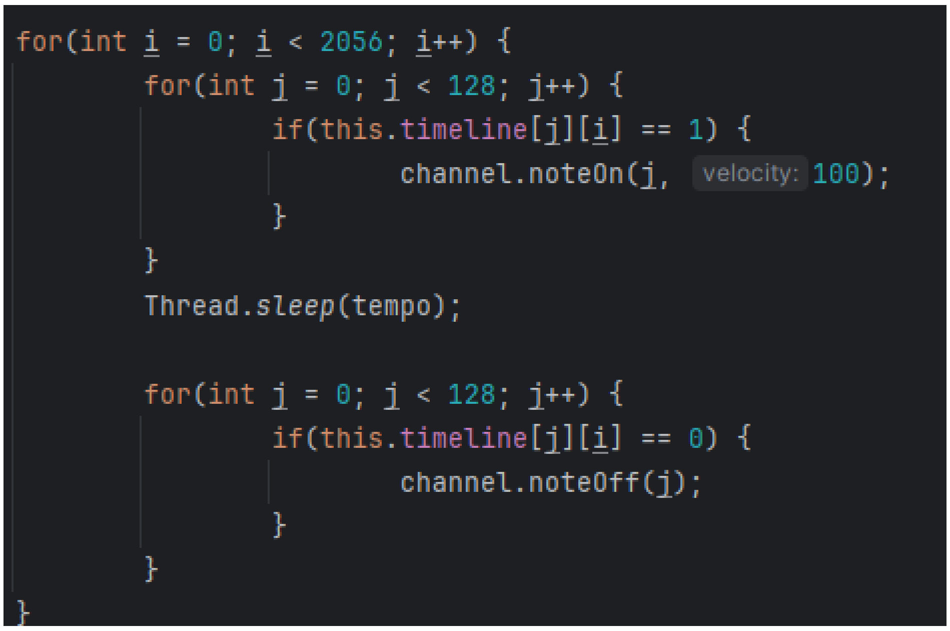
After that I added more features for the timeline, such as a button at the start of each line which the user could click to take them to that instrument's individual timeline, and JLabels for displaying each sequence. In addition to that I added a timelineProgressBar to display how far into the song 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
```

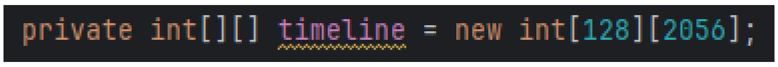
Before I started anything else, I tested my play method to make sure that the correct notes were playing in the correct order. I did this because it is fundamental to the rest of the program working. This is the code I ran:


```
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:

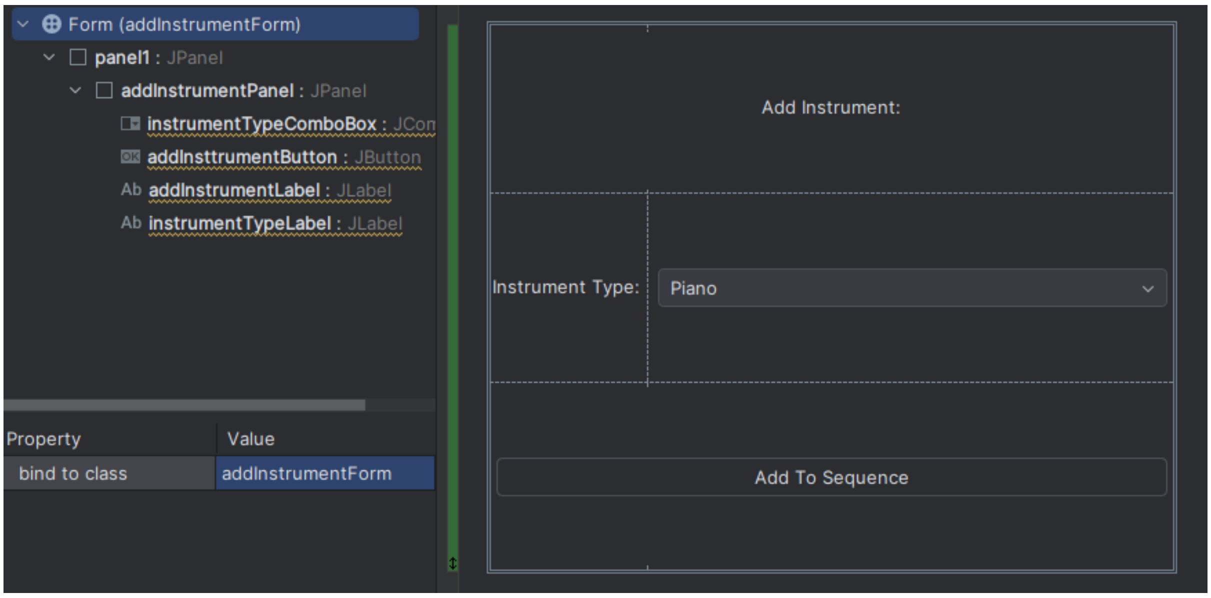


```java
for(int i = 0; i < 2056; i++) {
        for(int j = 0; j < 128; j++) {
                if(this.timeline[j][i] == 1) {
                        channel.noteOn(j, velocity: 100);
                }
        }
        Thread.sleep(tempo);

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



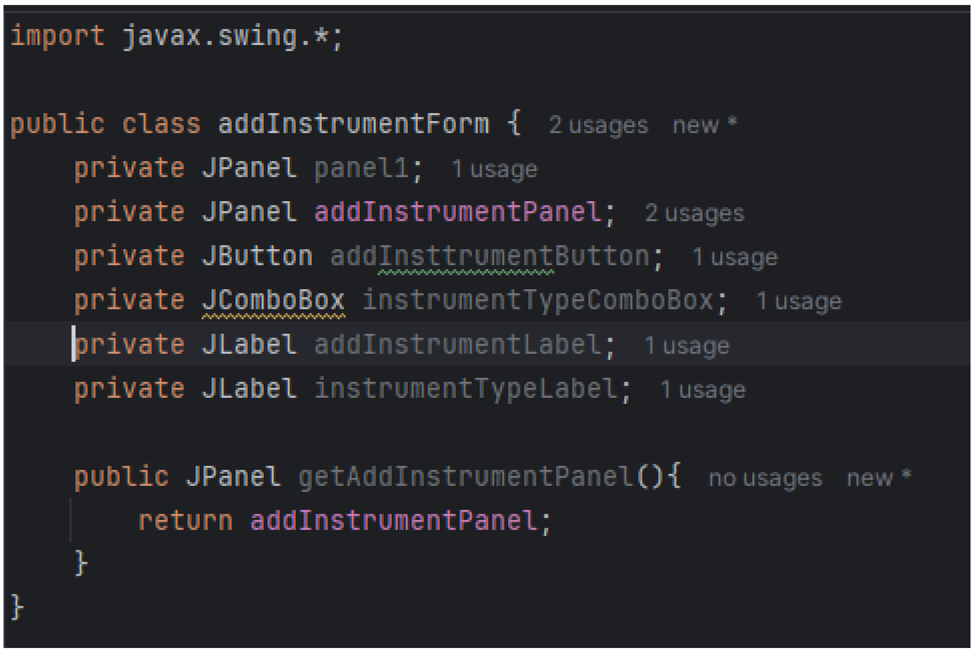
```java
private int[][] timeline = new int[128][2056];
```

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:



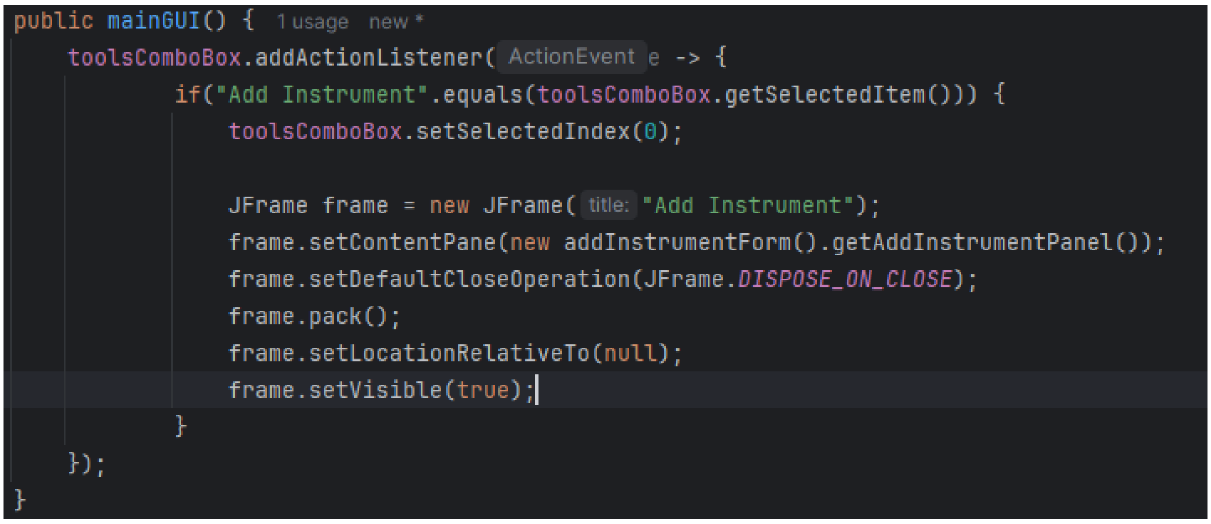
```
Form (addInstrumentForm)
  panel1 : JPanel
    addInstrumentPanel : JPanel
      instrumentTypeComboBox : JCon
      addInsttrumentButton : JButton
      addInstrumentLabel : JLabel
      instrumentTypeLabel : JLabel
```

Property	Value
bind to class	addInstrumentForm



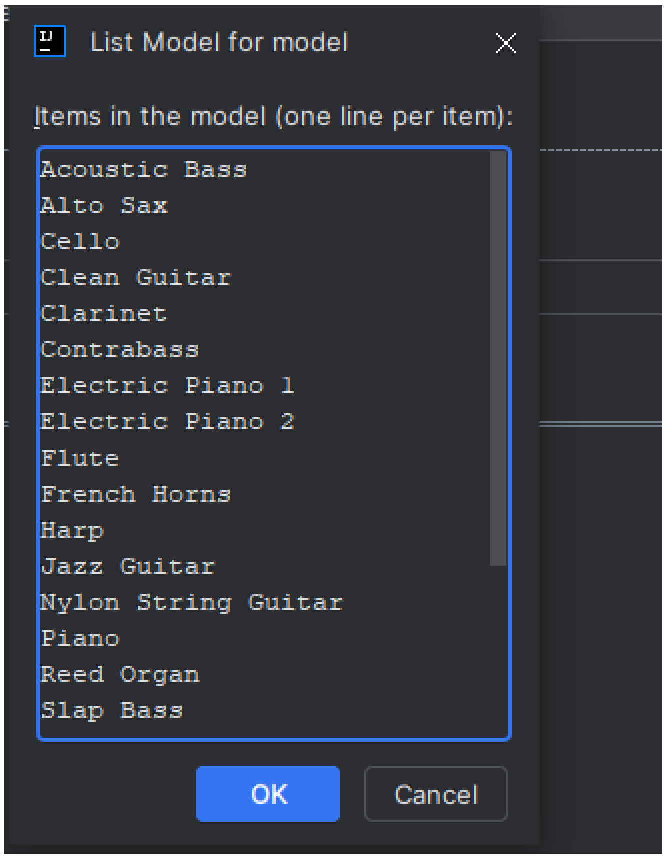
```
import javax.swing.*;

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:



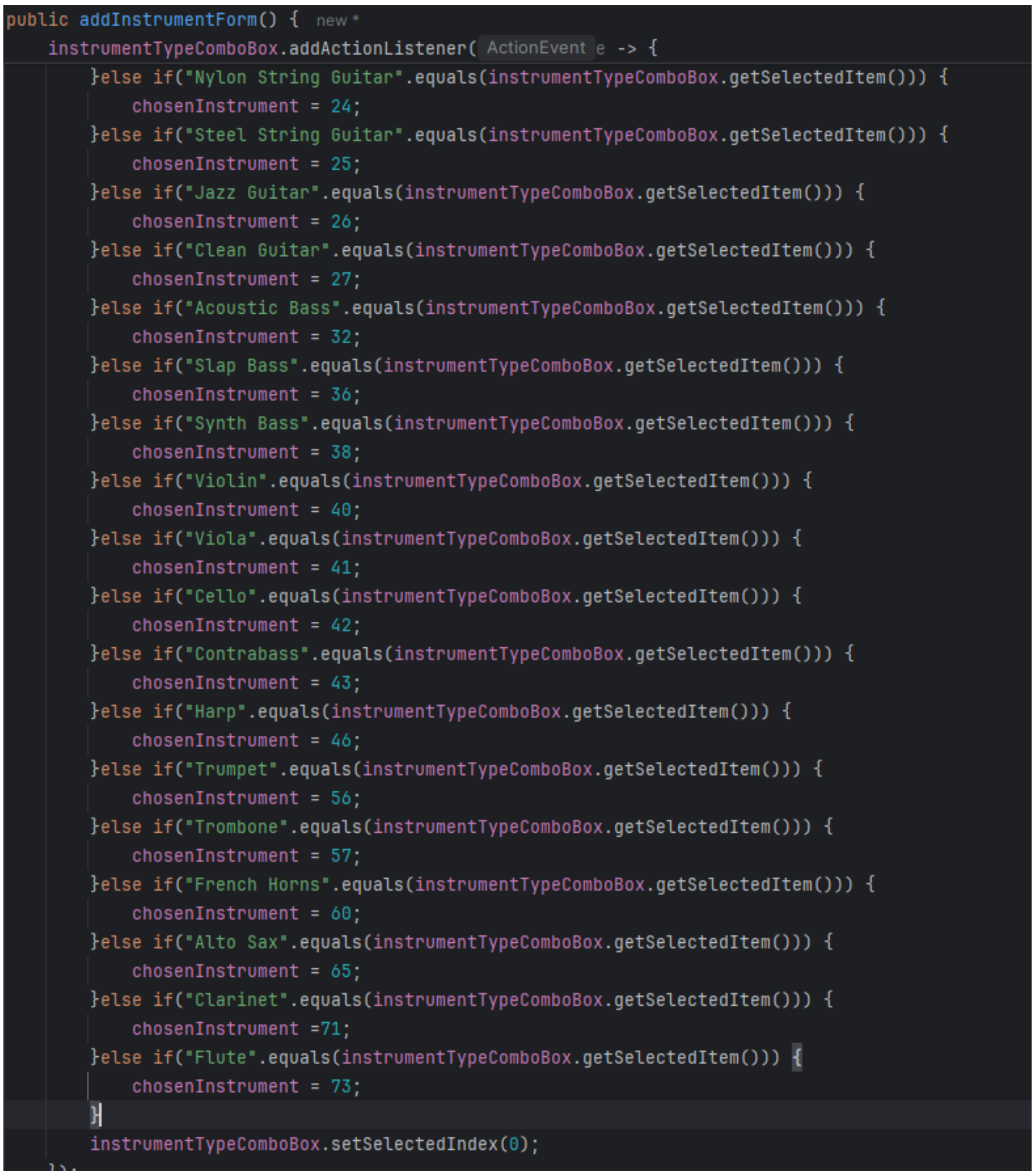
```
public mainGUI() {  1 usage  new *
    toolsComboBox.addActionListener( ActionEvent e -> {
            if("Add Instrument".equals(toolsComboBox.getSelectedItem())) {
                toolsComboBox.setSelectedIndex(0);

                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:



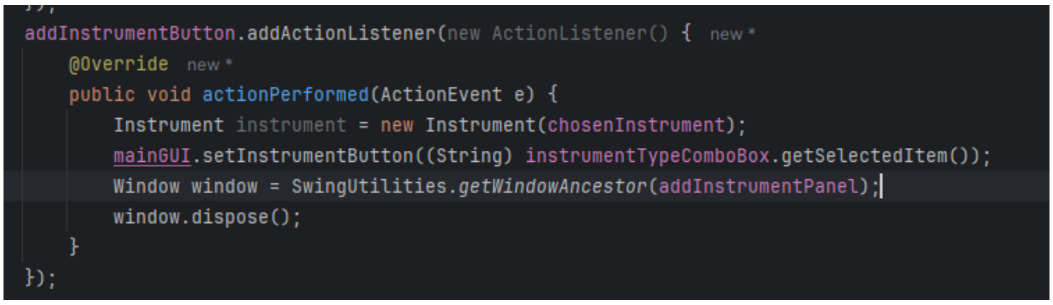
```
Acoustic Bass
Alto Sax
Cello
Clean Guitar
Clarinet
Contrabass
Electric Piano 1
Electric Piano 2
Flute
French Horns
Harp
Jazz Guitar
Nylon String Guitar
Piano
Reed Organ
Slap Bass
```

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:

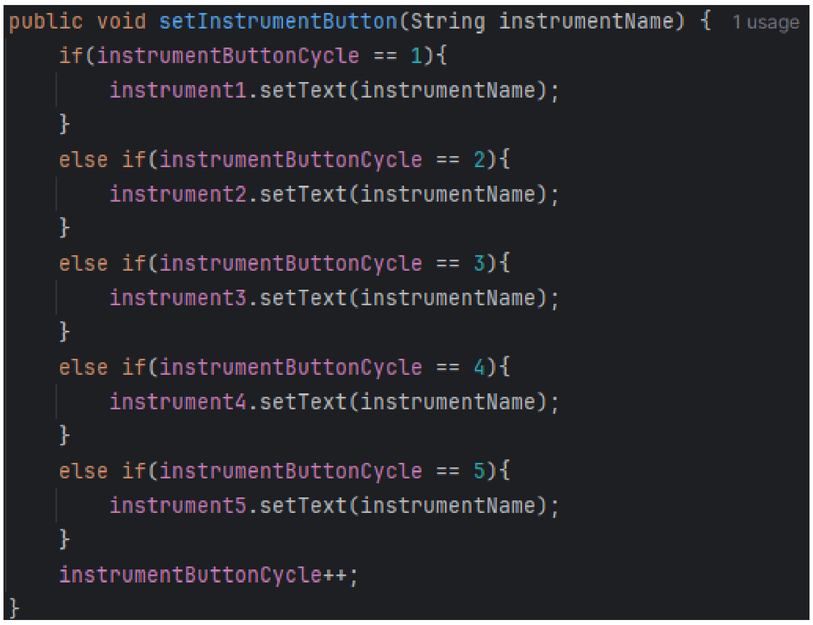


```
public addInstrumentForm() {  new *
    instrumentTypeComboBox.addActionListener( ActionEvent e -> {
        }else if("Nylon String Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 24;
        }else if("Steel String Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            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:

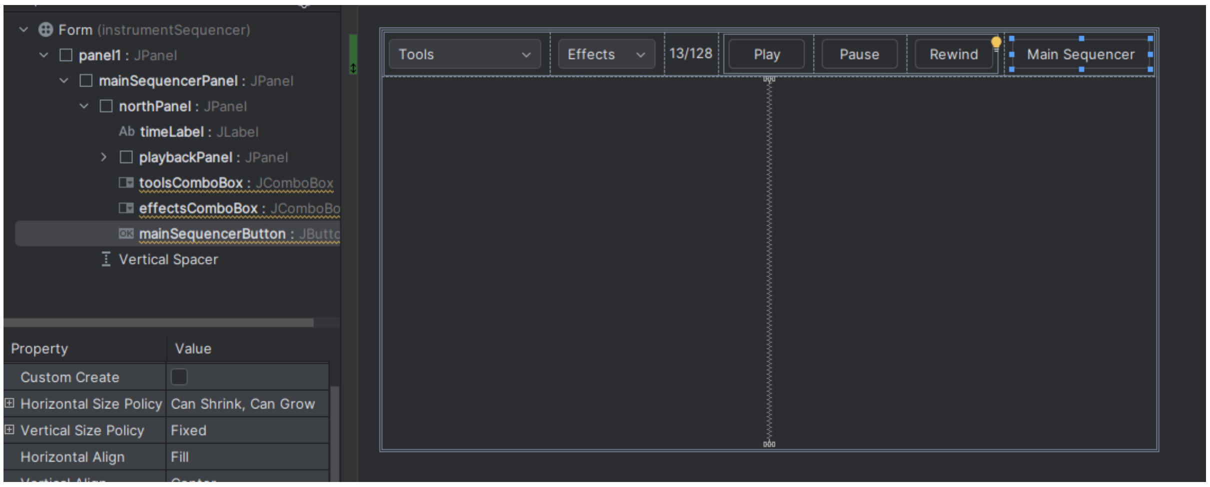


```
addInstrumentButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = new Instrument(chosenInstrument);
        mainGUI.setInstrumentButton((String) instrumentTypeComboBox.getSelectedItem());
    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



```java
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++;
}
```

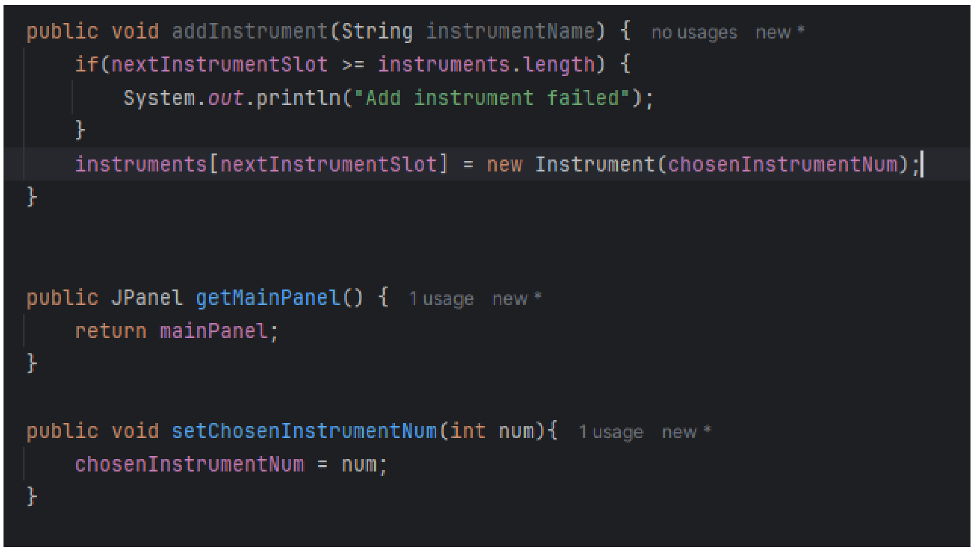
Next, I created another GUI form for the individual instrument sequencer called instrumentSequencer. I added Jbuttons, JComboBox's and a JLabel. The northPanel of this GUI is like that of the mainGUI, since they both contain sequencing and playback features. I created this GUI and the components within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



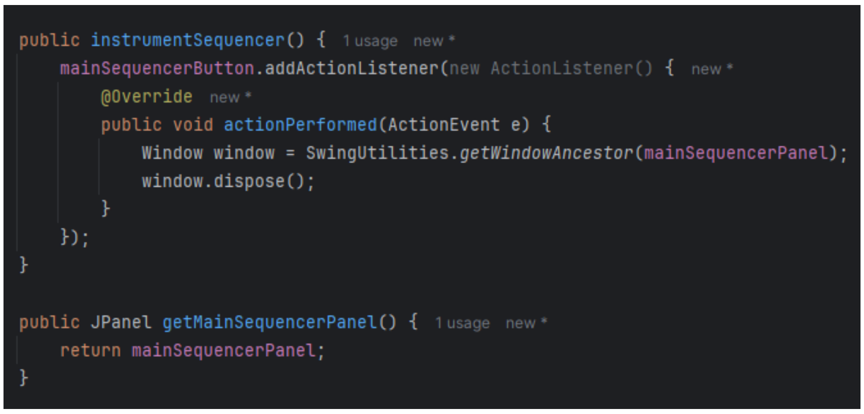
```
Form (instrumentSequencer)
  panel1 : JPanel
    mainSequencerPanel : JPanel
      northPanel : JPanel
        timeLabel : JLabel
        playbackPanel : JPanel
        toolsComboBox : JComboBox
        effectsComboBox : JComboBo
        mainSequencerButton : JButto
      Vertical Spacer
```

Property	Value
Custom Create	☐
Horizontal Size Policy	Can Shrink, Can Grow
Vertical Size Policy	Fixed
Horizontal Align	Fill

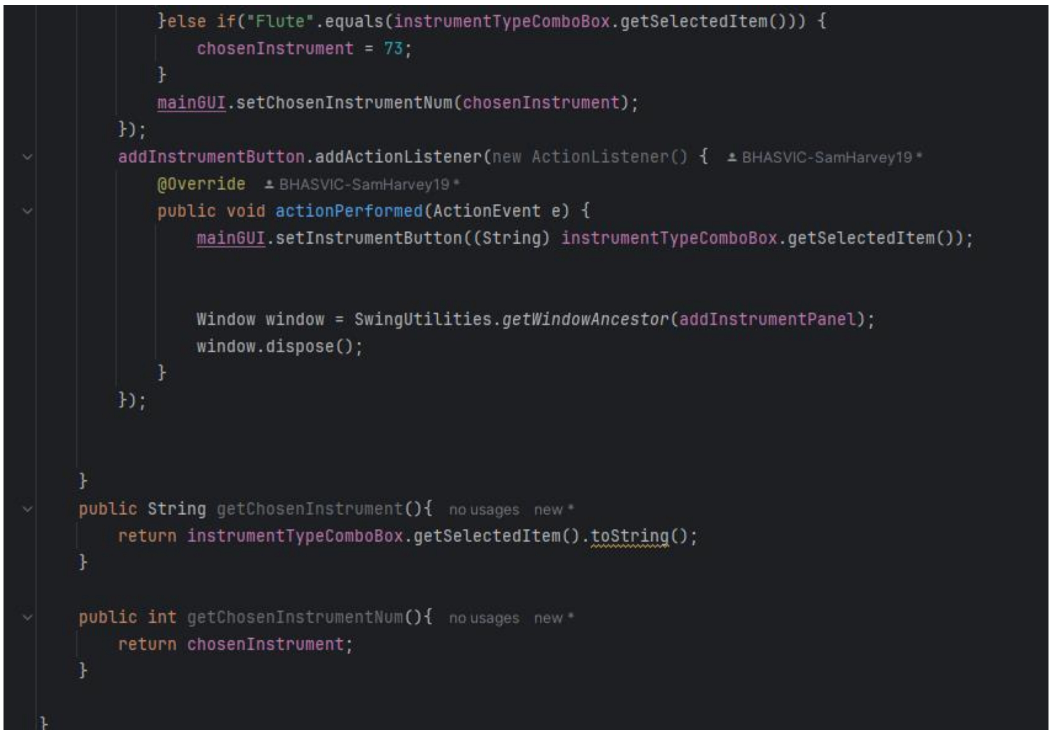
I also removed the tools and effects JComboBox from the GUI since they were not necessary for the user to use in this GUI. In addition, I added a JProgressBar so that the user can see how far along the playback they are, and a JScrollPane, which contains the sequencerSplitPlane, which further contains 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:



```
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;
}
```

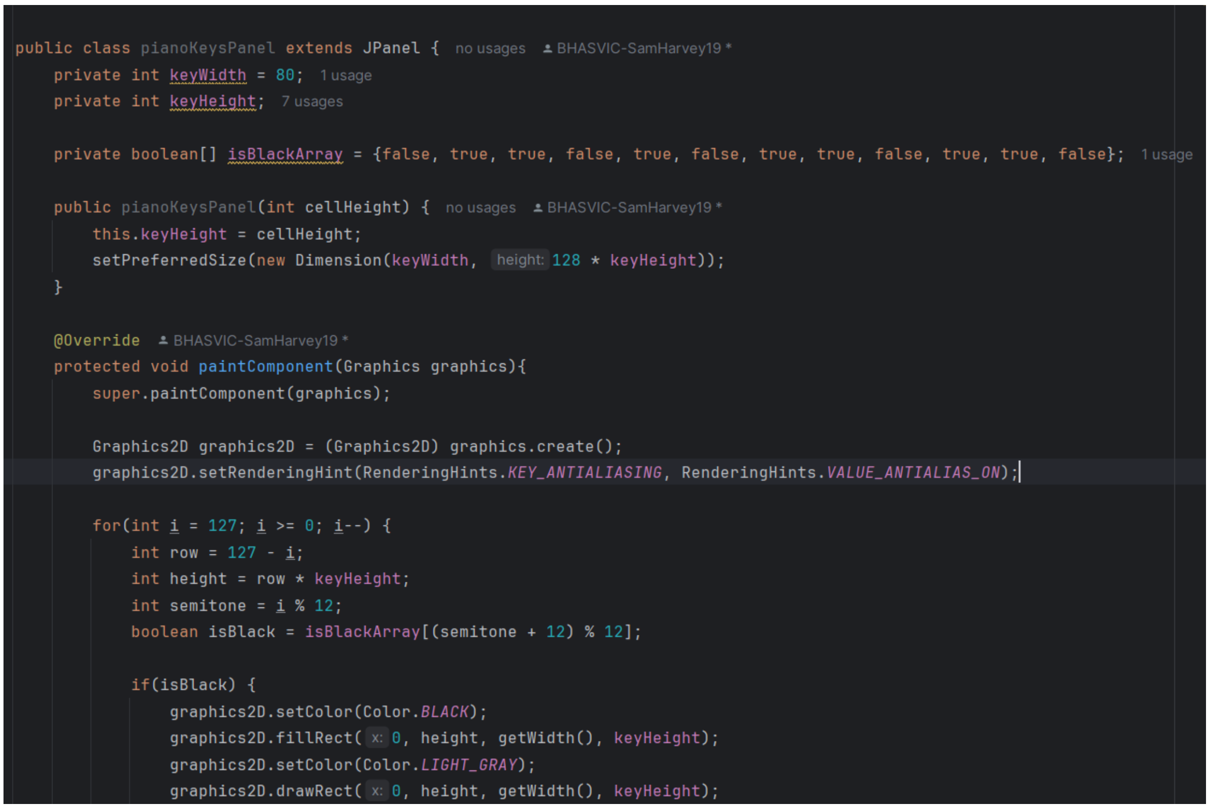


```
public instrumentSequencer() {  1 usage  new *
    mainSequencerButton.addActionListener(new ActionListener() {  new *
        @Override  new *
        public void actionPerformed(ActionEvent e) {
            Window window = SwingUtilities.getWindowAncestor(mainSequencerPanel);
            window.dispose();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

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

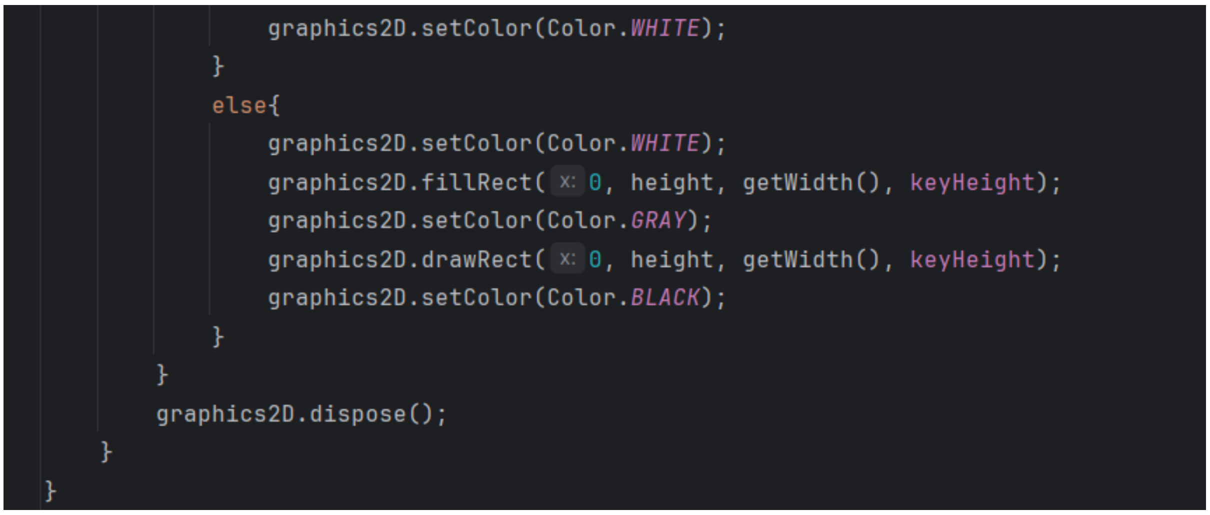


```
            }else if("Flute".equals(instrumentTypeComboBox.getSelectedItem())) {
                chosenInstrument = 73;
            }
            mainGUI.setChosenInstrumentNum(chosenInstrument);
        });
        addInstrumentButton.addActionListener(new ActionListener() {  BHASVIC-SamHarvey19 *
            @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:

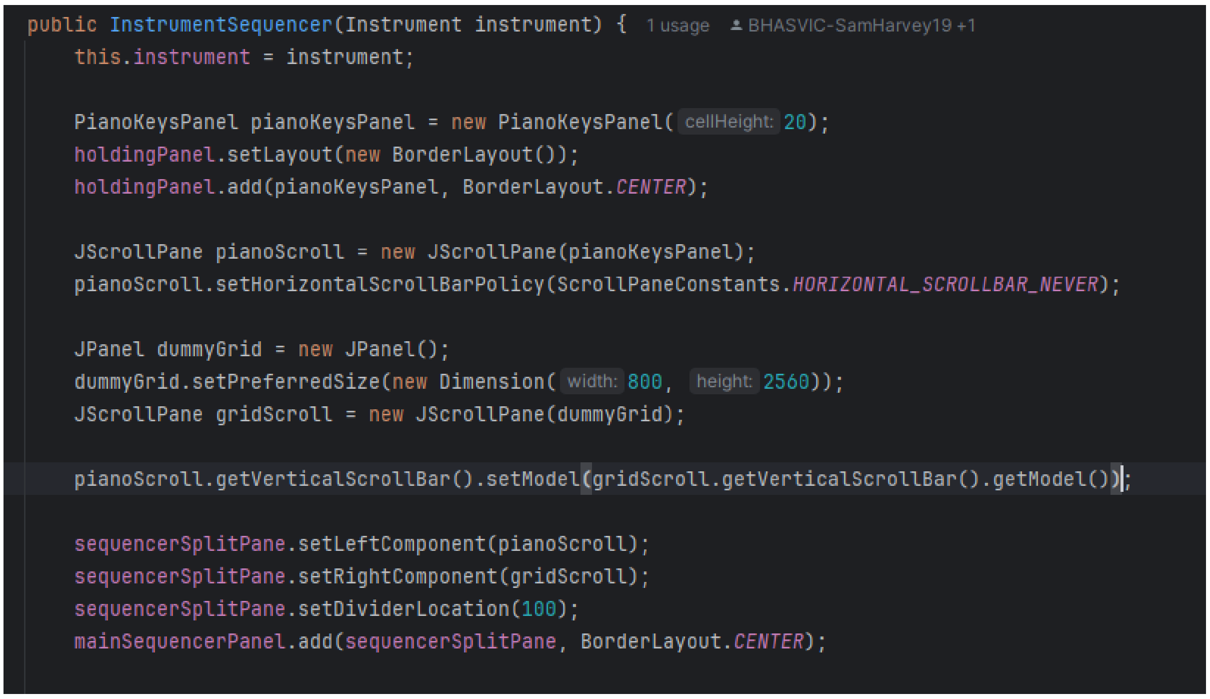


```java
public class pianoKeysPanel extends JPanel {  no usages  BHASVIC-SamHarvey19 *
    private int keyWidth = 80;  1 usage
    private int keyHeight;  7 usages

    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);
```

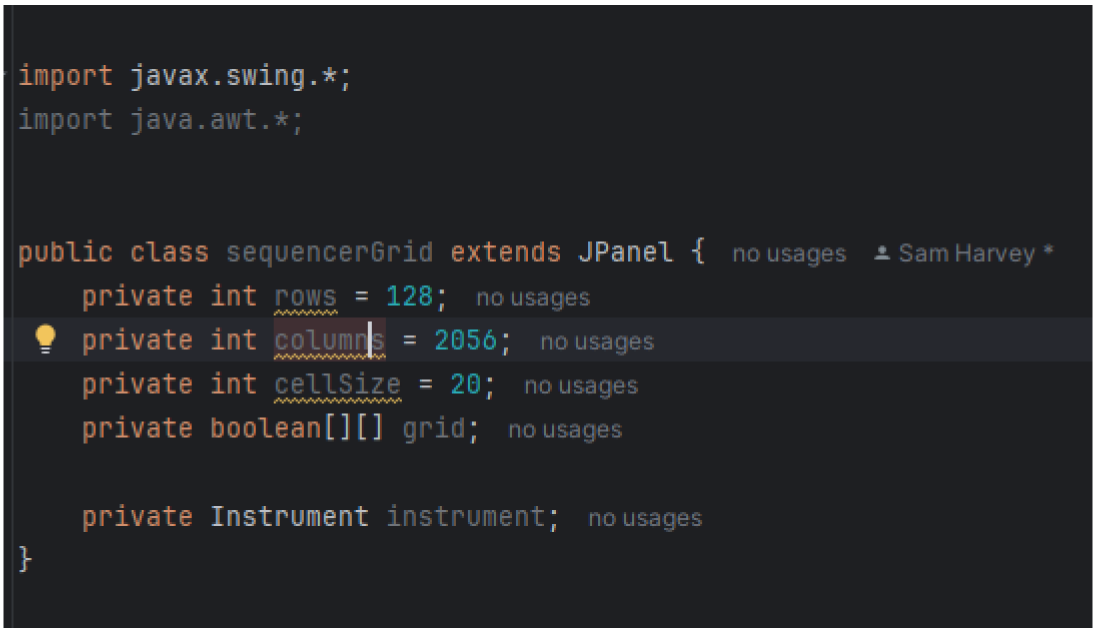


```java
                graphics2D.setColor(Color.WHITE);
            }
            else{
                graphics2D.setColor(Color.WHITE);
                graphics2D.fillRect( x: 0, height, getWidth(), keyHeight);
                graphics2D.setColor(Color.GRAY);
                graphics2D.drawRect(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

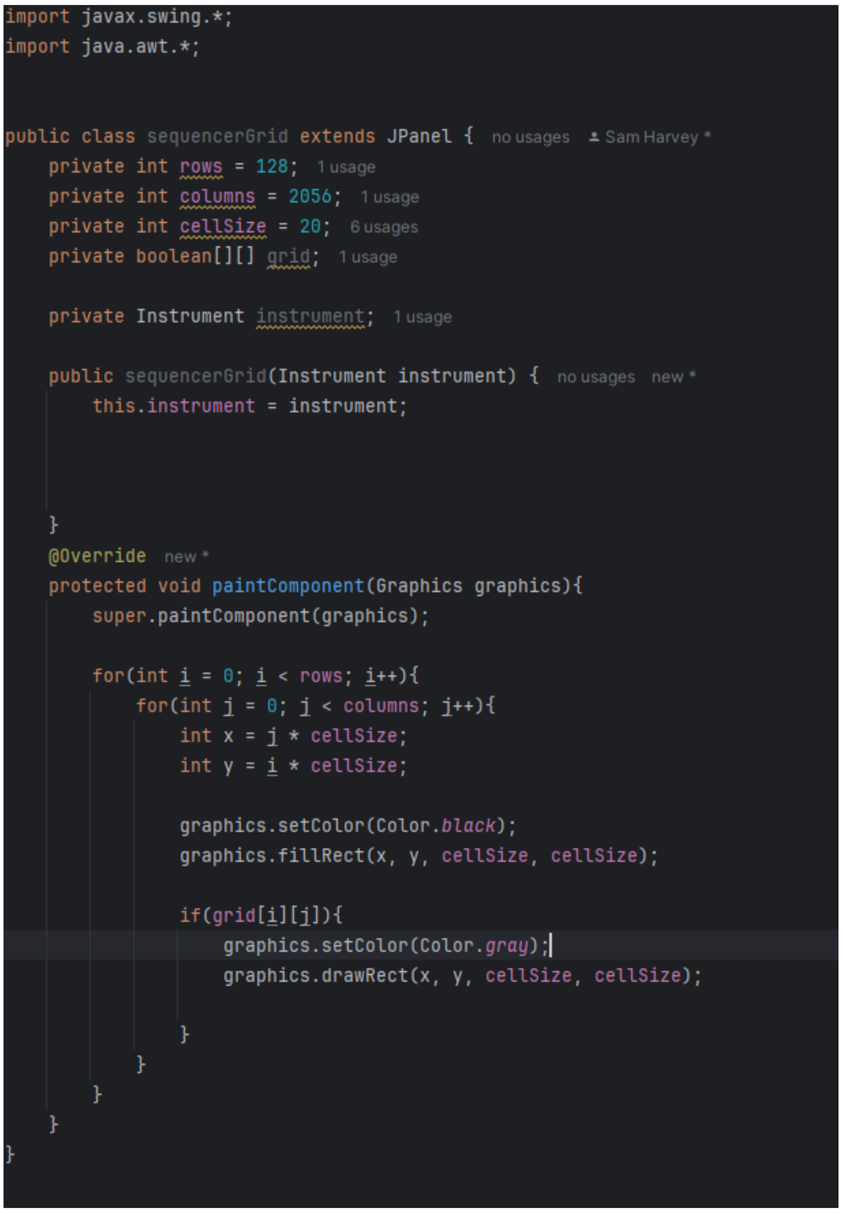

```java
public InstrumentSequencer(Instrument instrument) {  1 usage  BHASVIC-SamHarvey19 +1
    this.instrument = instrument;

    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:



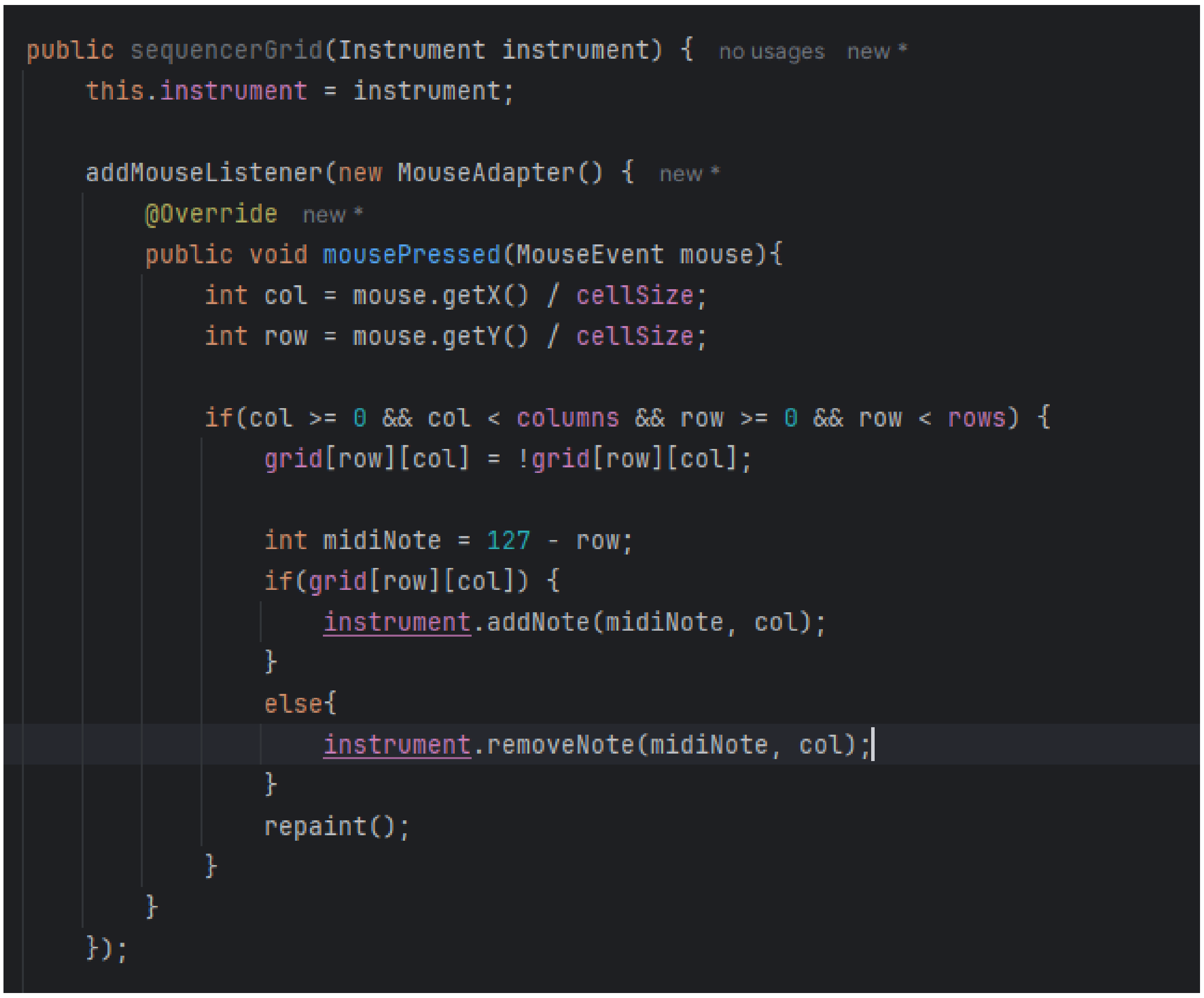
```java
import javax.swing.*;
import java.awt.*;


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:



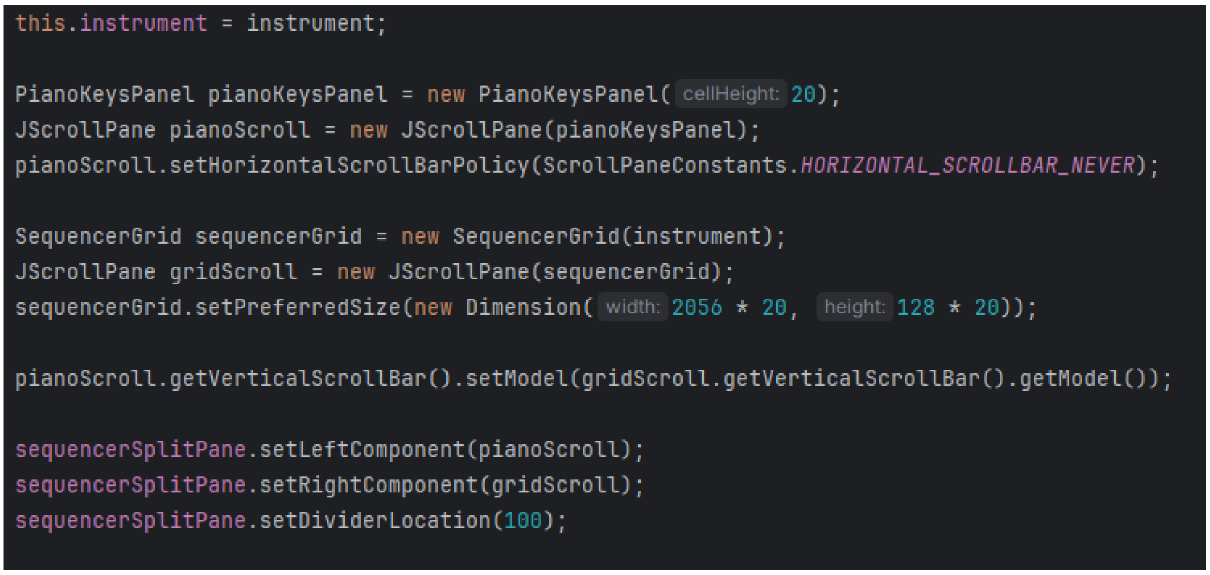
```
import javax.swing.*;
import java.awt.*;


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:



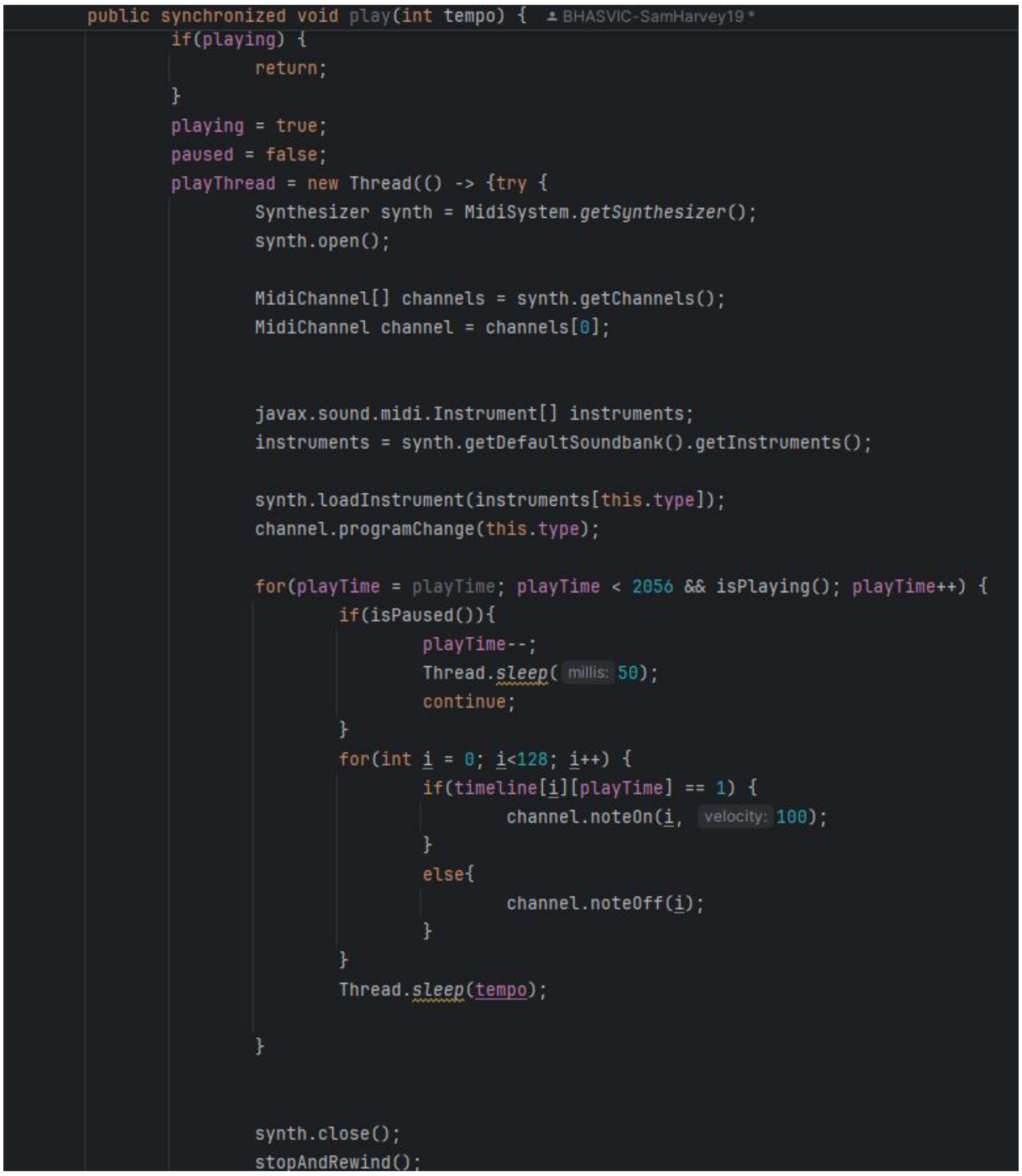
```
public sequencerGrid(Instrument instrument) {  no usages  new *
    this.instrument = instrument;

    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:

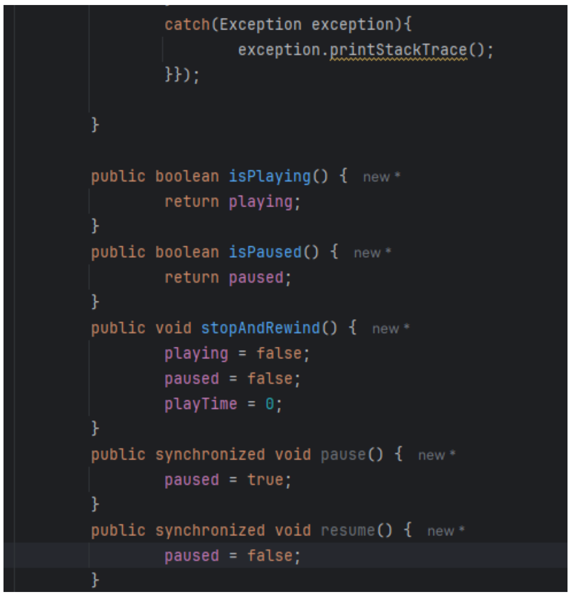


```java
this.instrument = instrument;

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:

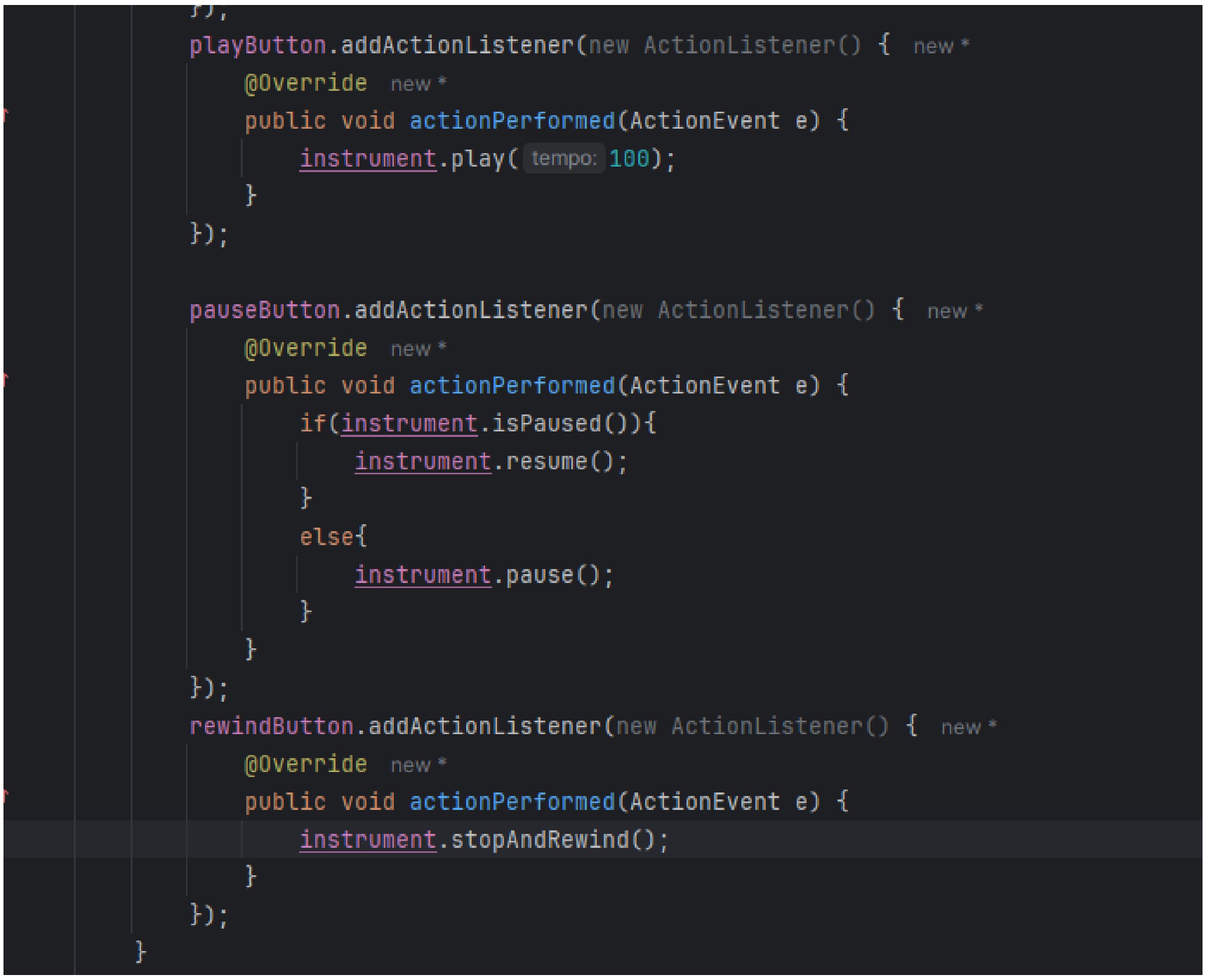


```
public synchronized void play(int tempo) {  BHASVIC-SamHarvey19 *
        if(playing) {
                return;
        }
        playing = true;
        paused = false;
        playThread = new Thread(() -> {try {
                Synthesizer synth = MidiSystem.getSynthesizer();
                synth.open();

                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();
```



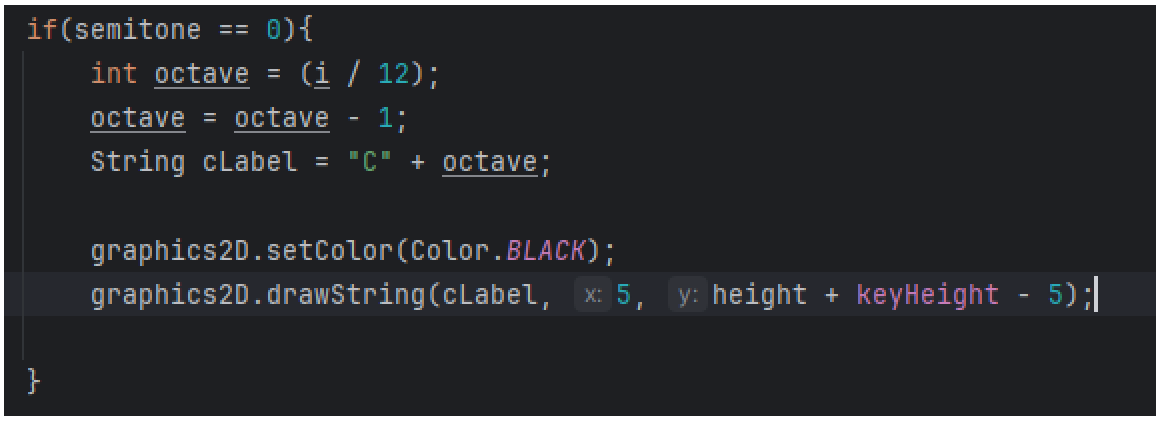
```
        catch(Exception exception){
                exception.printStackTrace();
        }});

}

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:



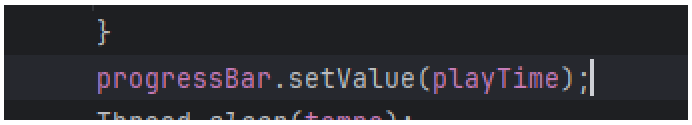
```
        });
        playButton.addActionListener(new ActionListener() {  new *
            @Override  new *
            public void actionPerformed(ActionEvent e) {
                instrument.play( tempo: 100);
            }
        });

        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:

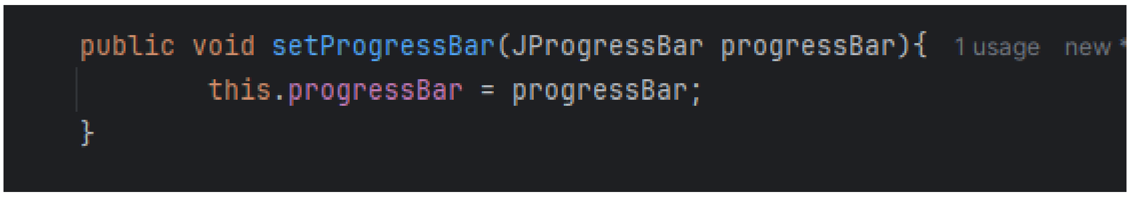


```
if(semitone == 0){
    int octave = (i / 12);
    octave = octave - 1;
    String cLabel = "C" + octave;

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

}
```

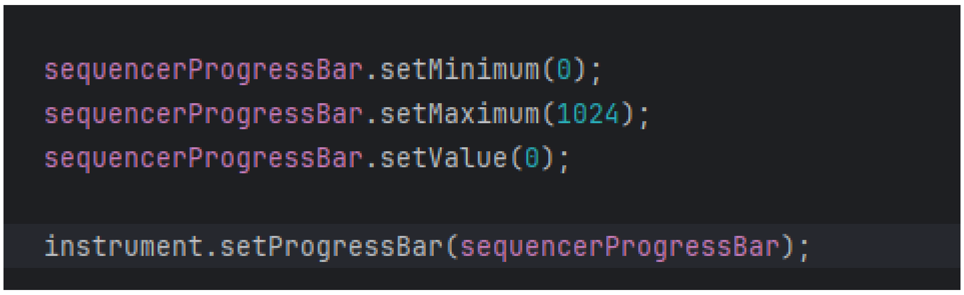
After that was completed, I set my focus back onto creating a moving bar that travels across the sequence as it is played. For this, I had to learn various methods that involve using JProgressBars (E.g. setValue(), setMinimum, setMaximum()). To learn this, I again referred to geeksforgeeks (https://www.geeksforgeeks.org/java/java-swing-jprogressbar/). 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);
Thread.sleep(tempo);
```

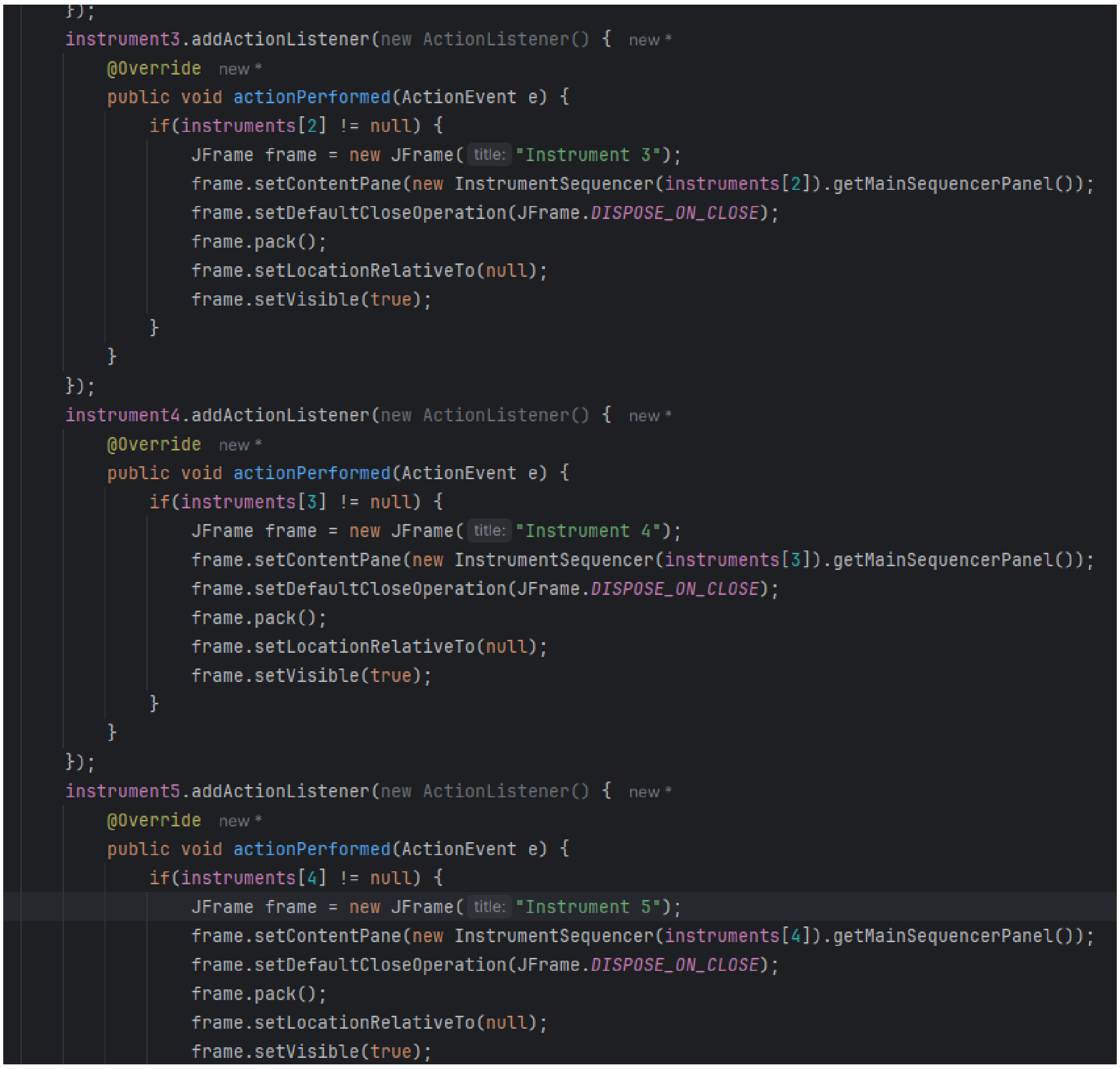


```
public void setProgressBar(JProgressBar progressBar){  1 usage  new *
        this.progressBar = progressBar;
}
```



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

instrument.setProgressBar(sequencerProgressBar);
```

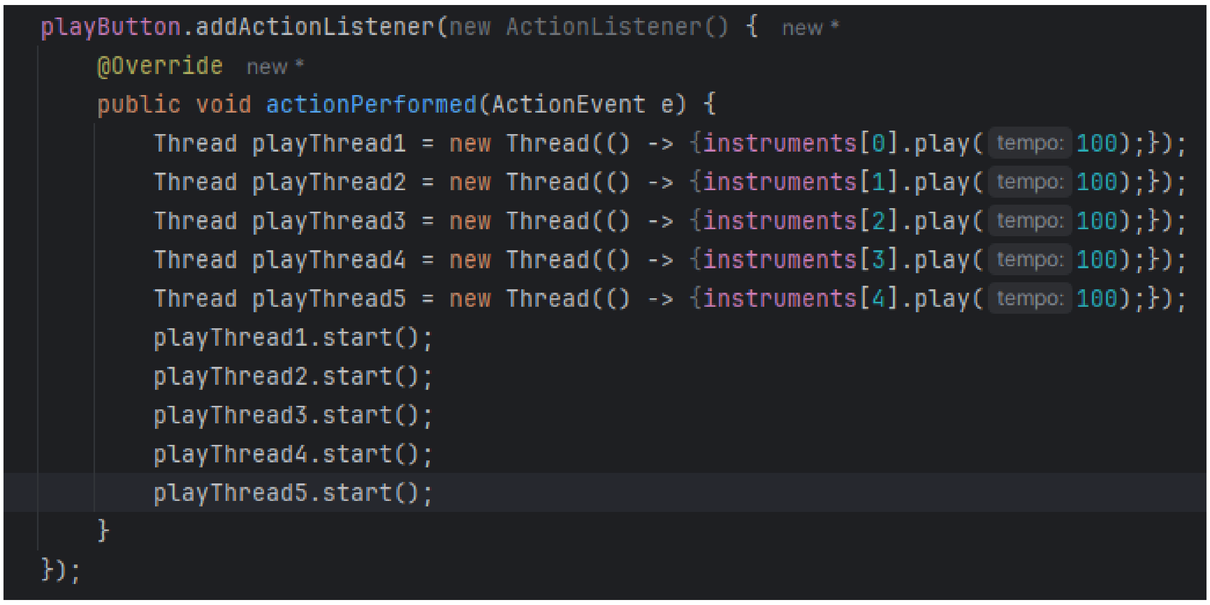
I tested this, and although the bar was slightly stuttered, it worked as expected. I expect this stutter was due to the power of the computer that I was coding on, so I did not see this as a problem. I am almost done with coding stage 1, however I did need to make it so that more than 1 instrument was 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);
```

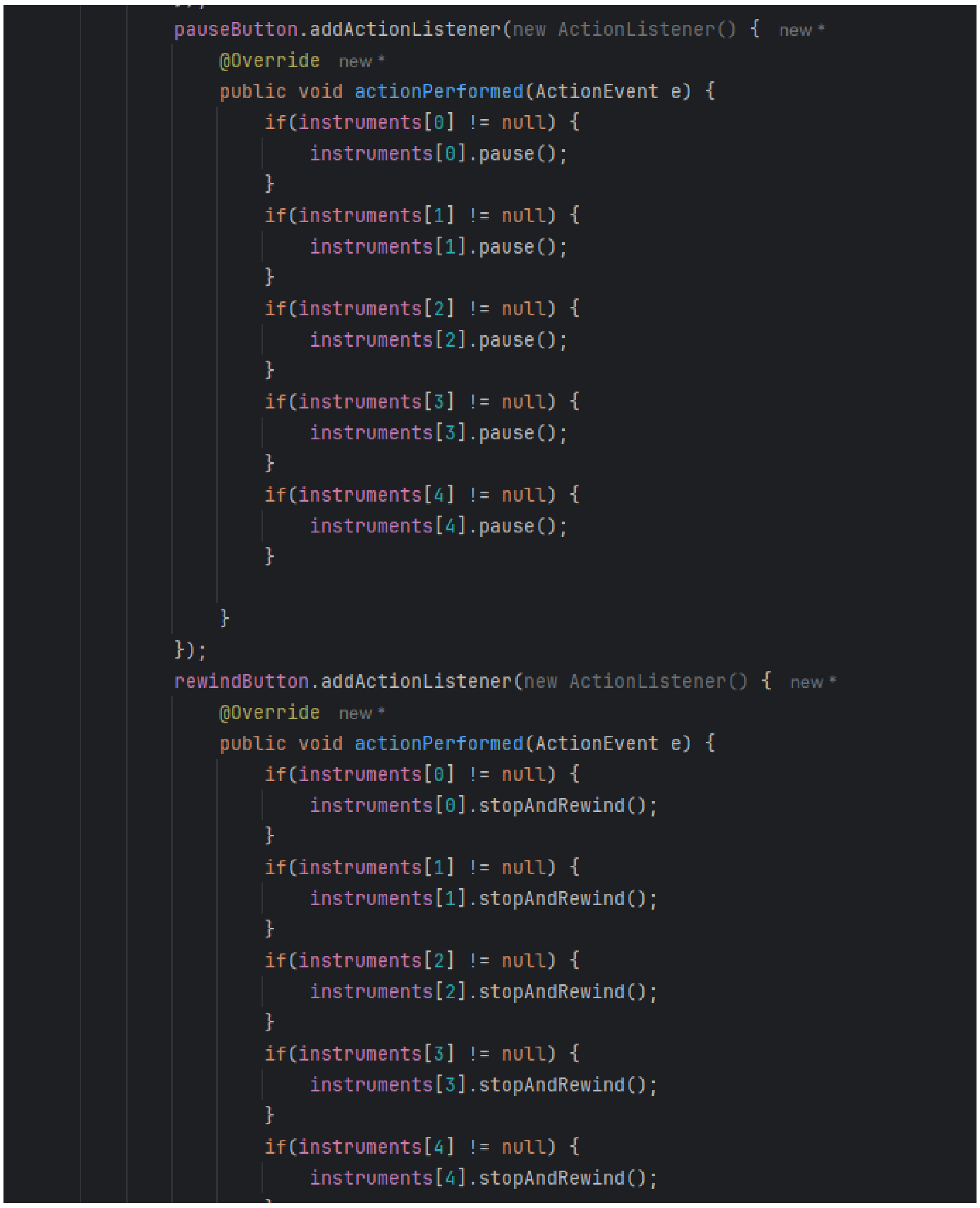
Next, I had to make the instrument1Timeline be a visual of the notes that are actually sequenced on that instrument. I did this not only for instrument1Timeline, but for all instrumentTimelines that are on the mainGUI. I did this by setting the contents of each of these JPanels to that of the SequencerGrid 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

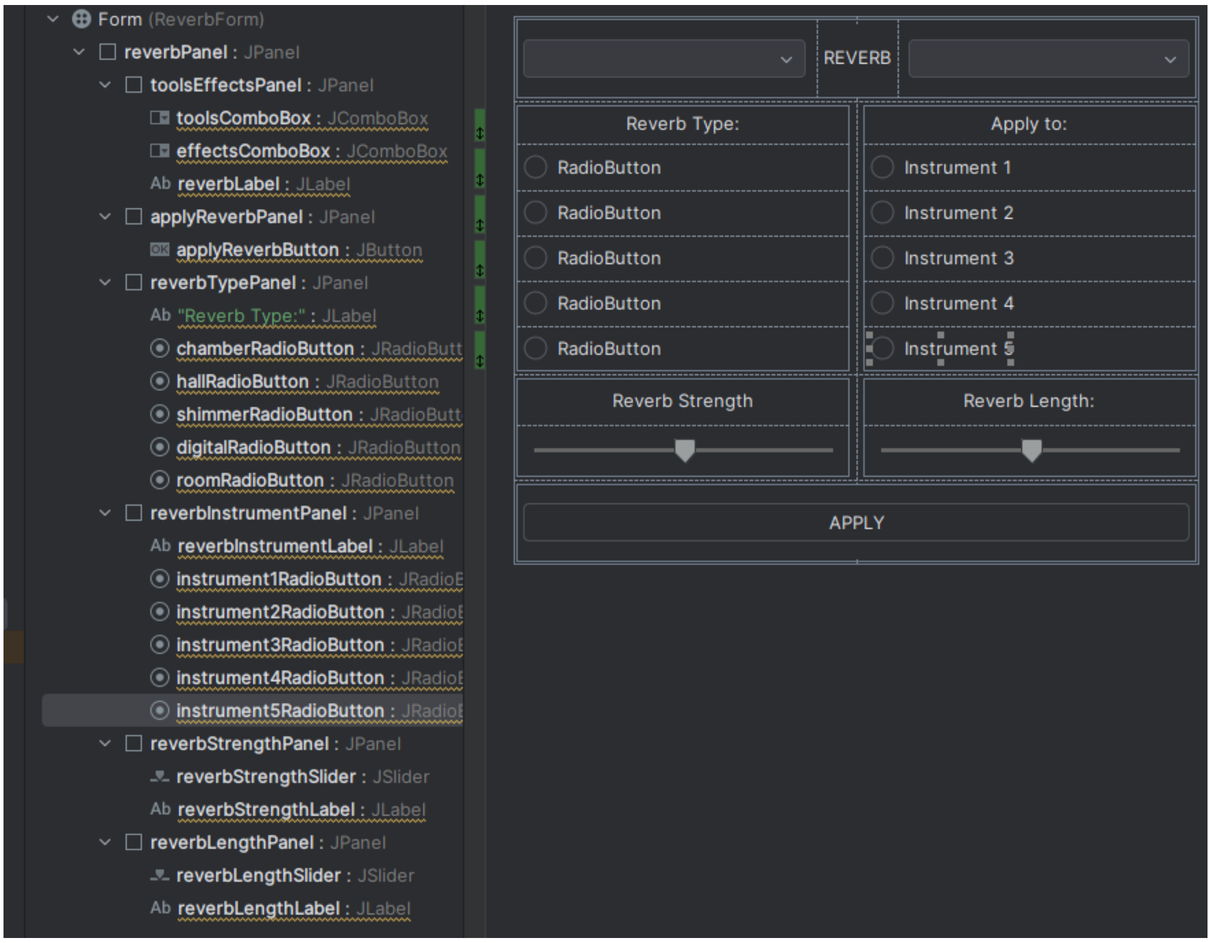
Methods:

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

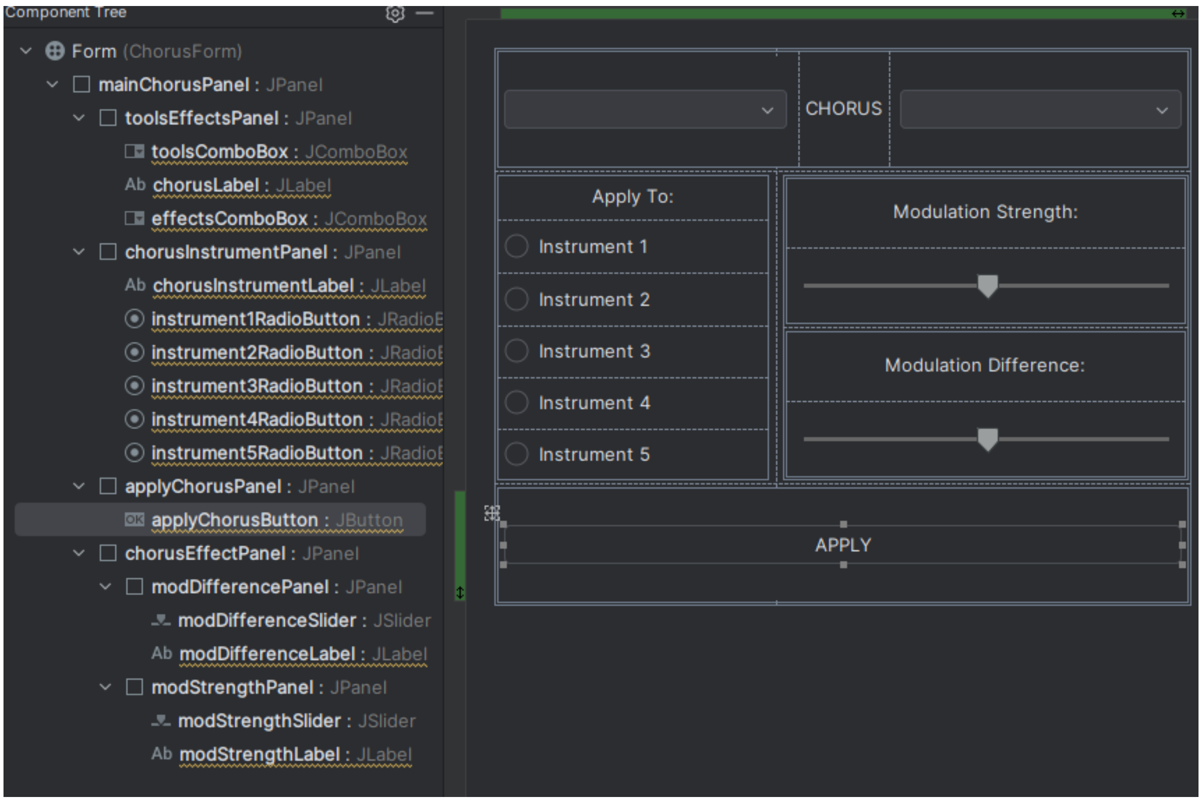
<u>Development:</u>

- 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:



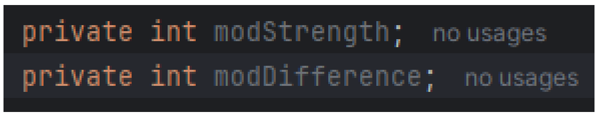
```
Form (ReverbForm)
  reverbPanel : JPanel
    toolsEffectsPanel : JPanel
      toolsComboBox : JComboBox
      effectsComboBox : JComboBox
      reverbLabel : JLabel
    applyReverbPanel : JPanel
      applyReverbButton : JButton
    reverbTypePanel : JPanel
      "Reverb Type:" : JLabel
      chamberRadioButton : JRadioButt
      hallRadioButton : JRadioButton
      shimmerRadioButton : JRadioButt
      digitalRadioButton : JRadioButton
      roomRadioButton : JRadioButton
    reverbInstrumentPanel : JPanel
      reverbInstrumentLabel : JLabel
      instrument1RadioButton : JRadioE
      instrument2RadioButton : JRadioE
      instrument3RadioButton : JRadioE
      instrument4RadioButton : JRadioE
      instrument5RadioButton : JRadioE
    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:

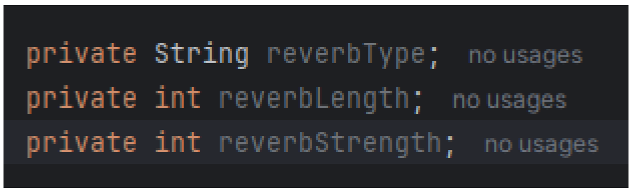


```
Component Tree
Form (ChorusForm)
  mainChorusPanel : JPanel
    toolsEffectsPanel : JPanel
      toolsComboBox : JComboBox
      chorusLabel : JLabel
      effectsComboBox : JComboBox
    chorusInstrumentPanel : JPanel
      chorusInstrumentLabel : JLabel
      instrument1RadioButton : JRadioB
      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:

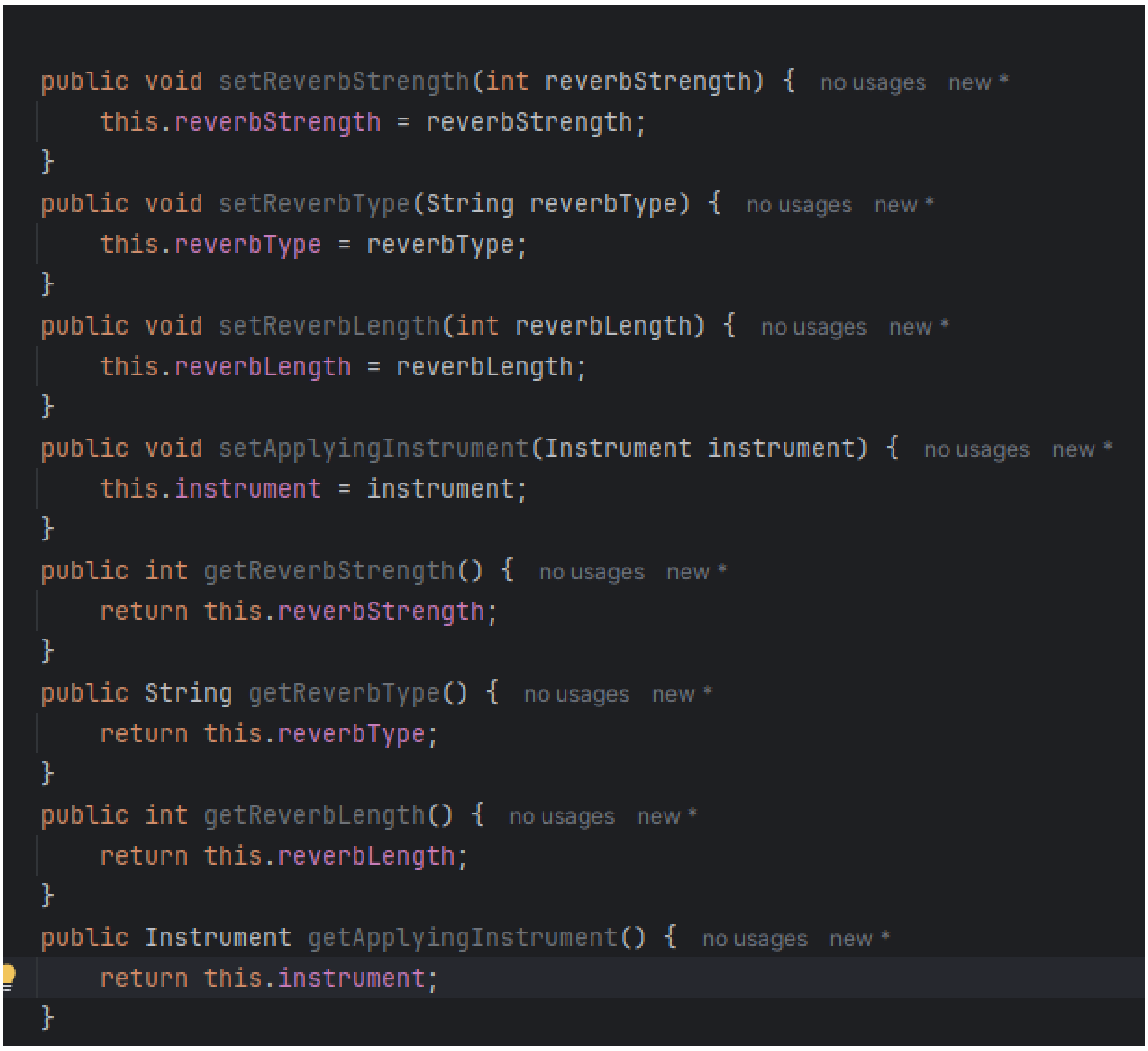


```
private int modStrength;  no usages
private int modDifference;  no usages
```



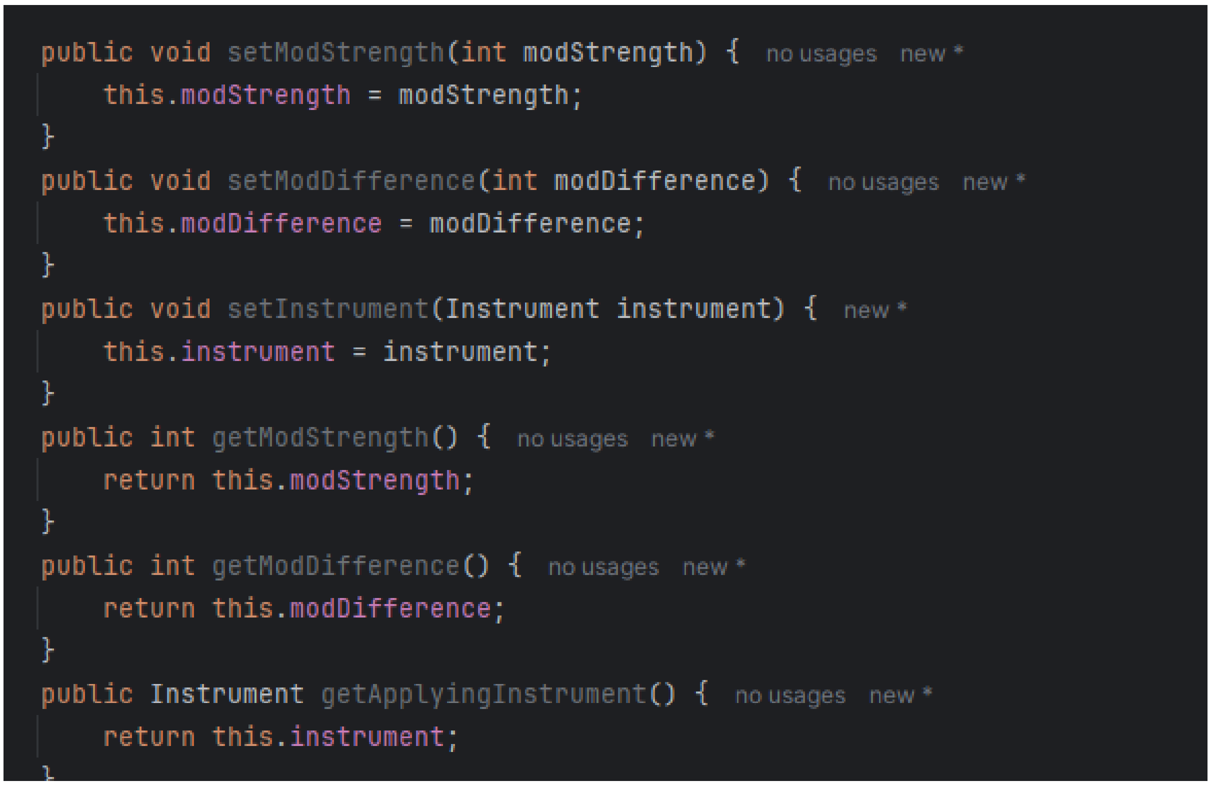
```
private String reverbType;  no usages
private int reverbLength;  no usages
private int reverbStrength;  no usages
```

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:



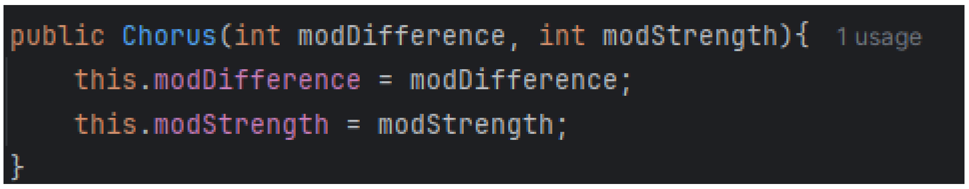
```
public void setReverbStrength(int reverbStrength) {  no usages   new *
    this.reverbStrength = reverbStrength;
}
public void setReverbType(String reverbType) {  no usages   new *
    this.reverbType = reverbType;
}
public void setReverbLength(int reverbLength) {  no usages   new *
    this.reverbLength = 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:

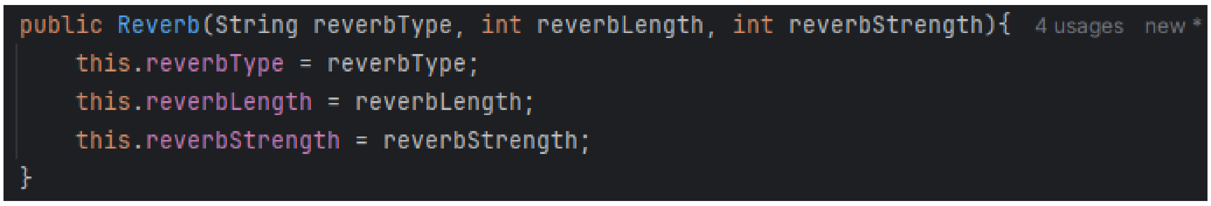


```java
public void setModStrength(int modStrength) {  no usages   new *
    this.modStrength = modStrength;
}
public void setModDifference(int modDifference) {  no usages   new *
    this.modDifference = modDifference;
}
public void setInstrument(Instrument instrument) {  new *
    this.instrument = 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:

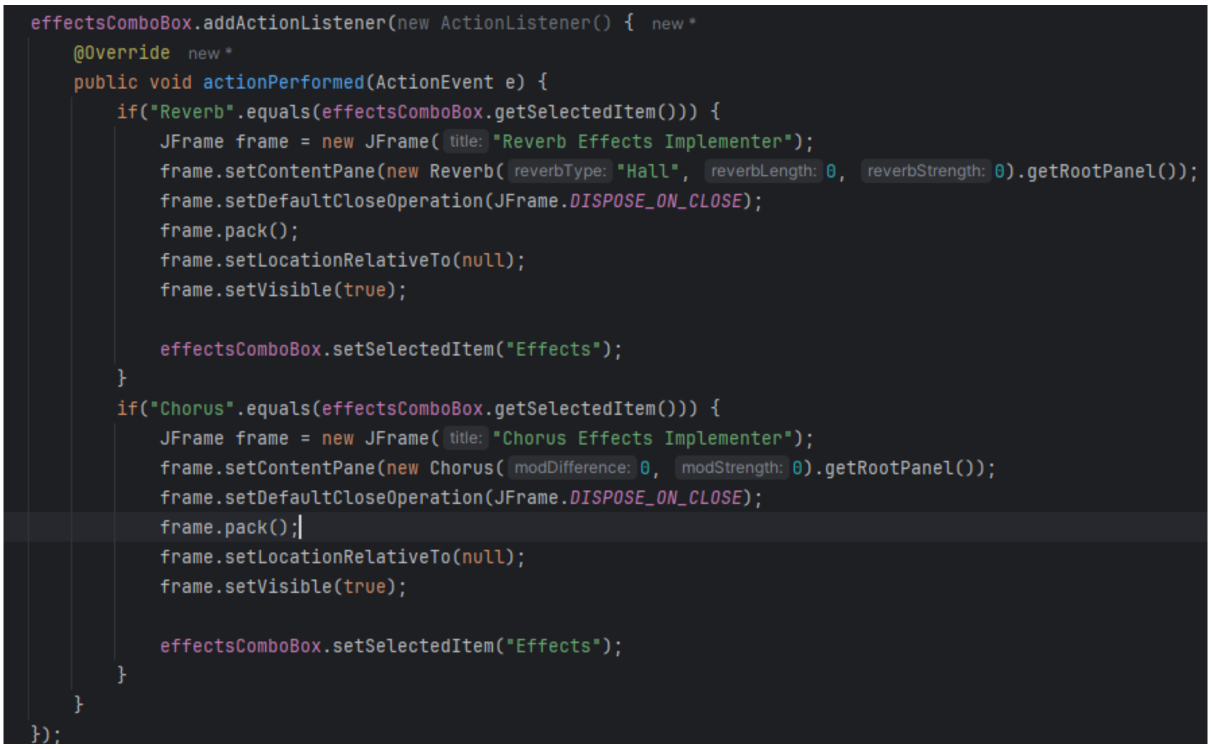


```java
public Chorus(int modDifference, int modStrength){  1 usage
    this.modDifference = modDifference;
    this.modStrength = modStrength;
}
```



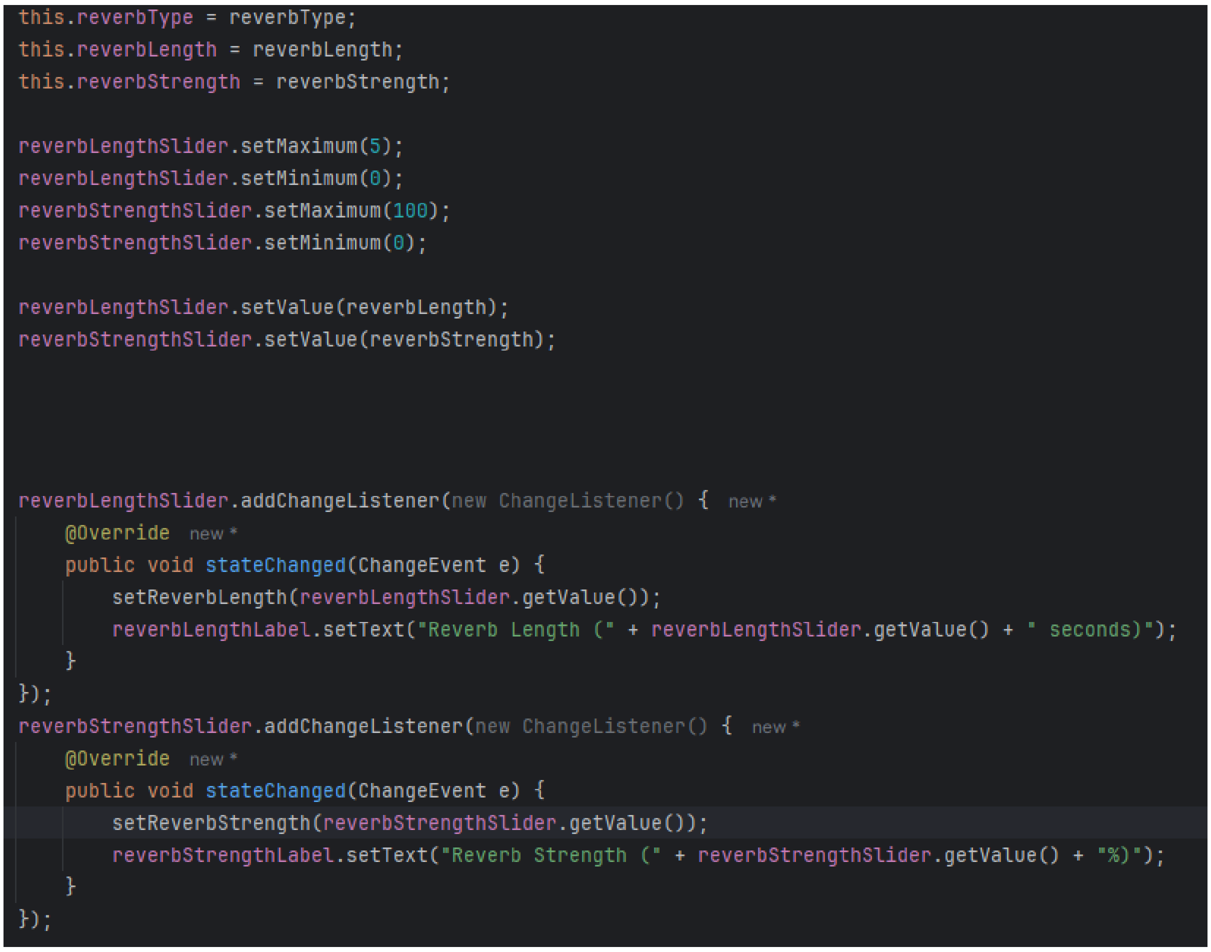
```java
public Reverb(String reverbType, int reverbLength, int reverbStrength){  4 usages   new *
    this.reverbType = reverbType;
    this.reverbLength = reverbLength;
    this.reverbStrength = reverbStrength;
}
```

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:



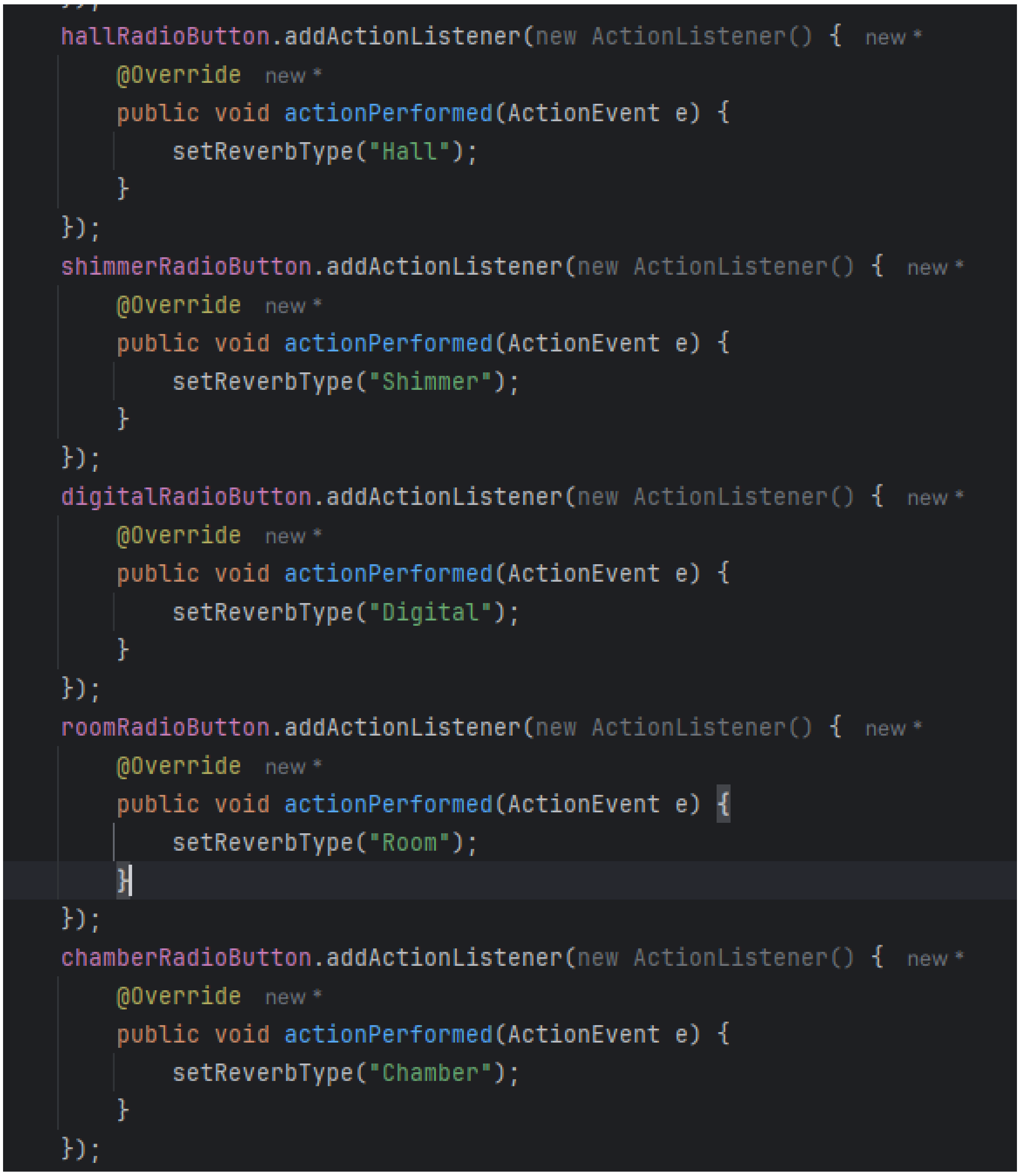
```
effectsComboBox.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        if("Reverb".equals(effectsComboBox.getSelectedItem())) {
            JFrame frame = new JFrame( title: "Reverb Effects Implementer");
            frame.setContentPane(new 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:



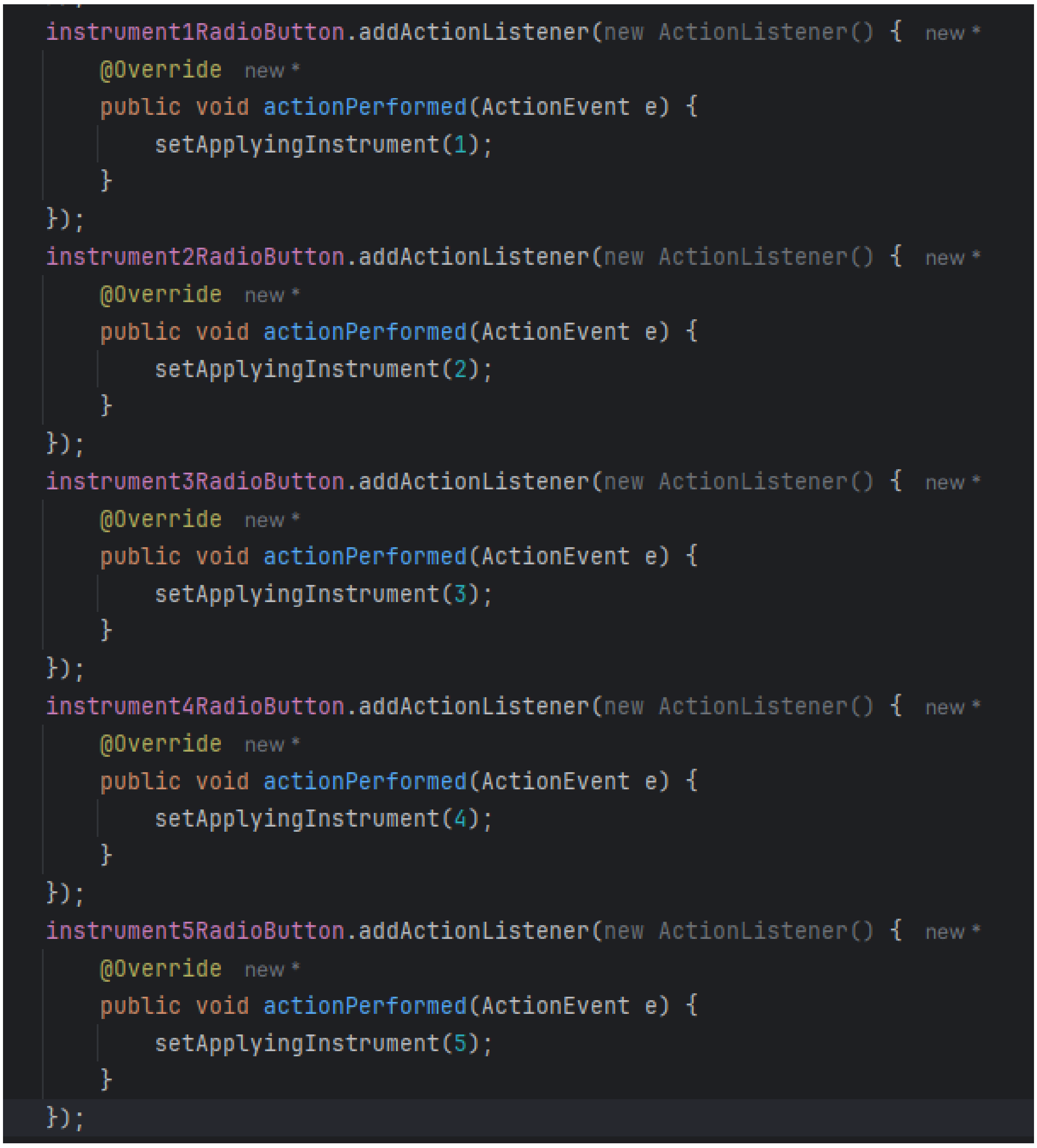
```
this.reverbType = reverbType;
this.reverbLength = reverbLength;
this.reverbStrength = reverbStrength;

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");
    }
});
```

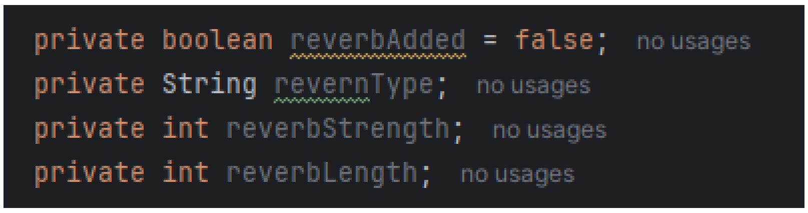
As you can see this is relatively simple code, as I only must call the setReverbType method to change the attribute reverbType with respect to what button has been selected. I used this same methodology to set the value of the applying instrument so that the user can change what instrument they would 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);
    }
});
```

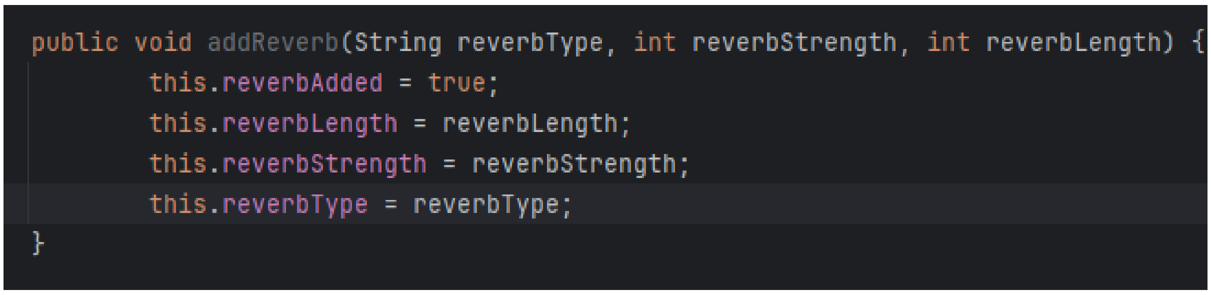
This code looks like the last block of code, by adding an action listener for each radio button, which sets the number of the applying instrument to a number relating to the index of the instrument.

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:



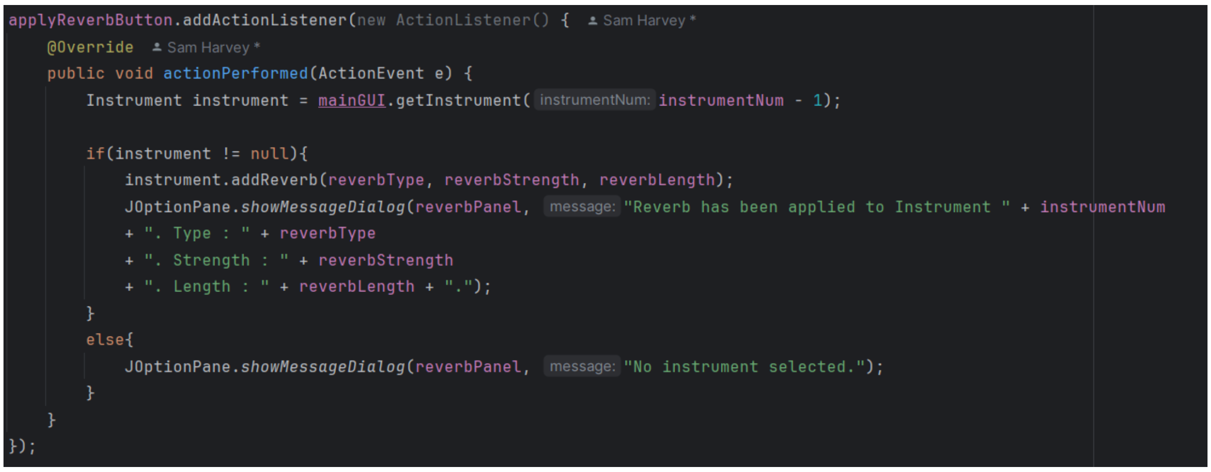
```
private boolean reverbAdded = false;  no usages
private String revernType;  no usages
private int reverbStrength;  no usages
private int reverbLength;  no usages
```

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:



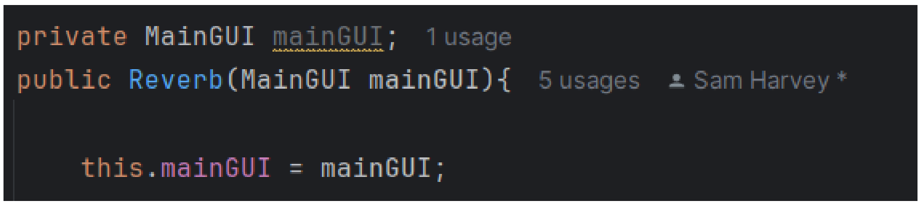
```
public void addReverb(String reverbType, int reverbStrength, int reverbLength) {
        this.reverbAdded = true;
        this.reverbLength = reverbLength;
        this.reverbStrength = reverbStrength;
        this.reverbType = reverbType;
}
```

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.");
        }
    }
});
```

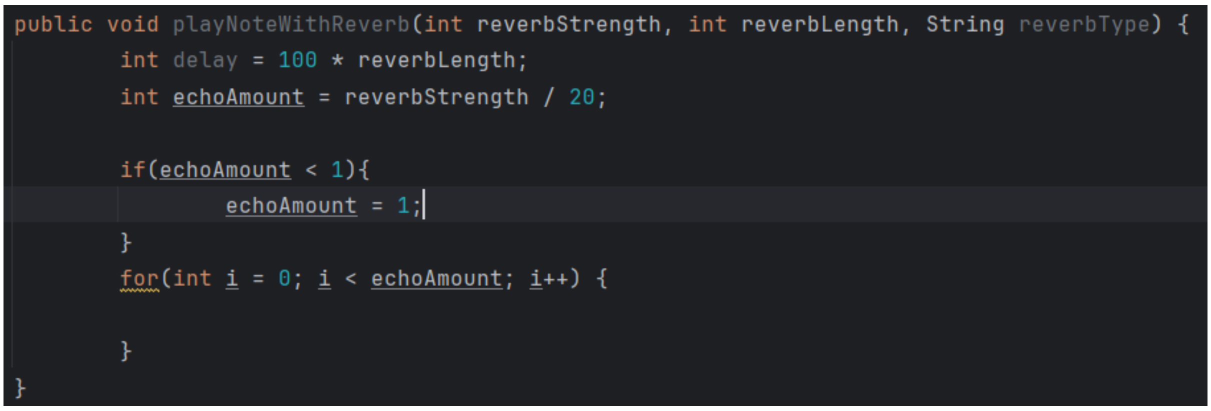
I tested this and it did not work. I then realised that I had not declared the mainGUI so I was not able to run the addReverb method. I went back into my constructor for the reverb class and changed. I did this so that I could run the addReverb method and therefore apply reverb to an instrument. This 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;
```

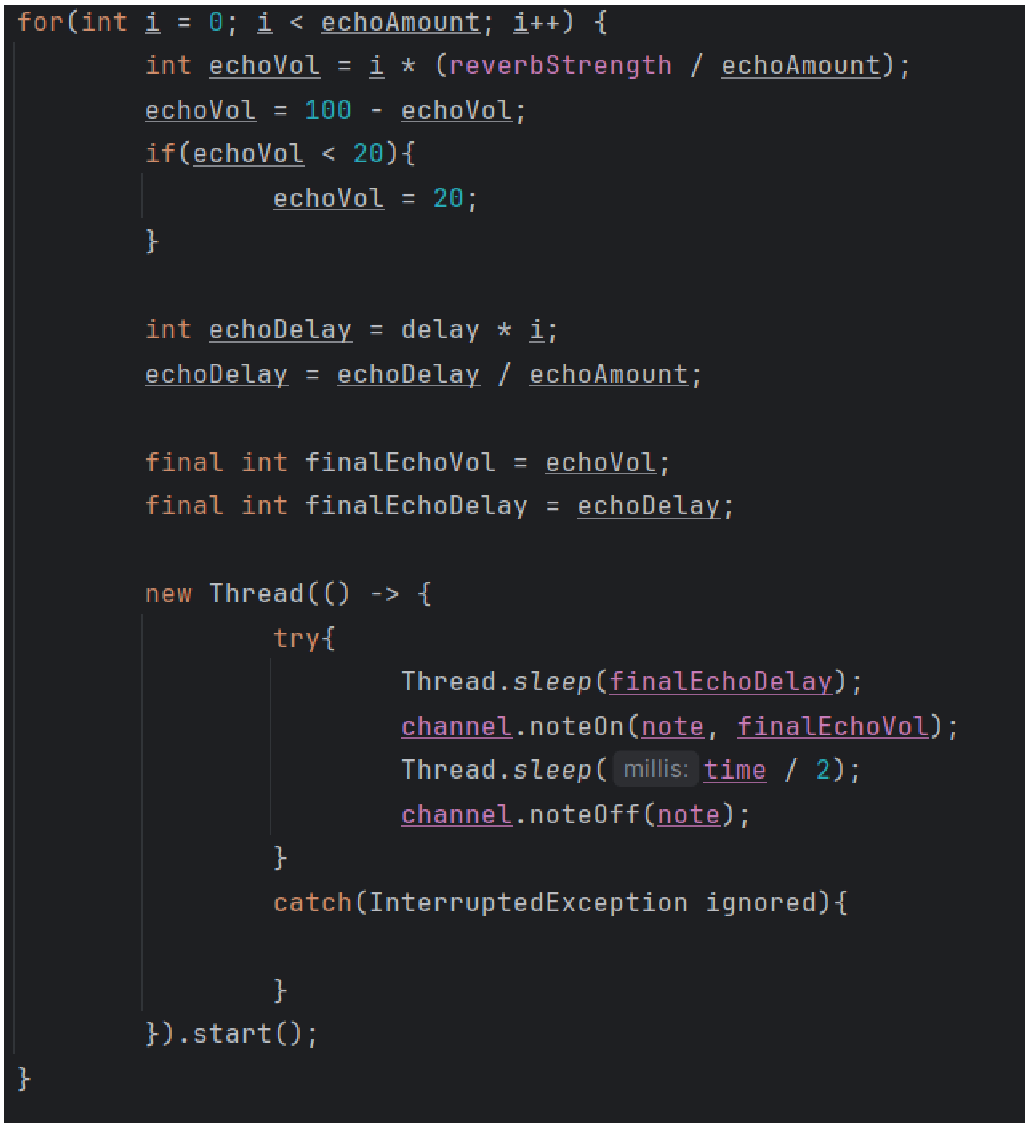
I tested this again and it worked (i.e. the JOptionPane showed up with a message saying the reverb had been added to the desired instrument with desired attributes). This implies that reverb has been added.

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:



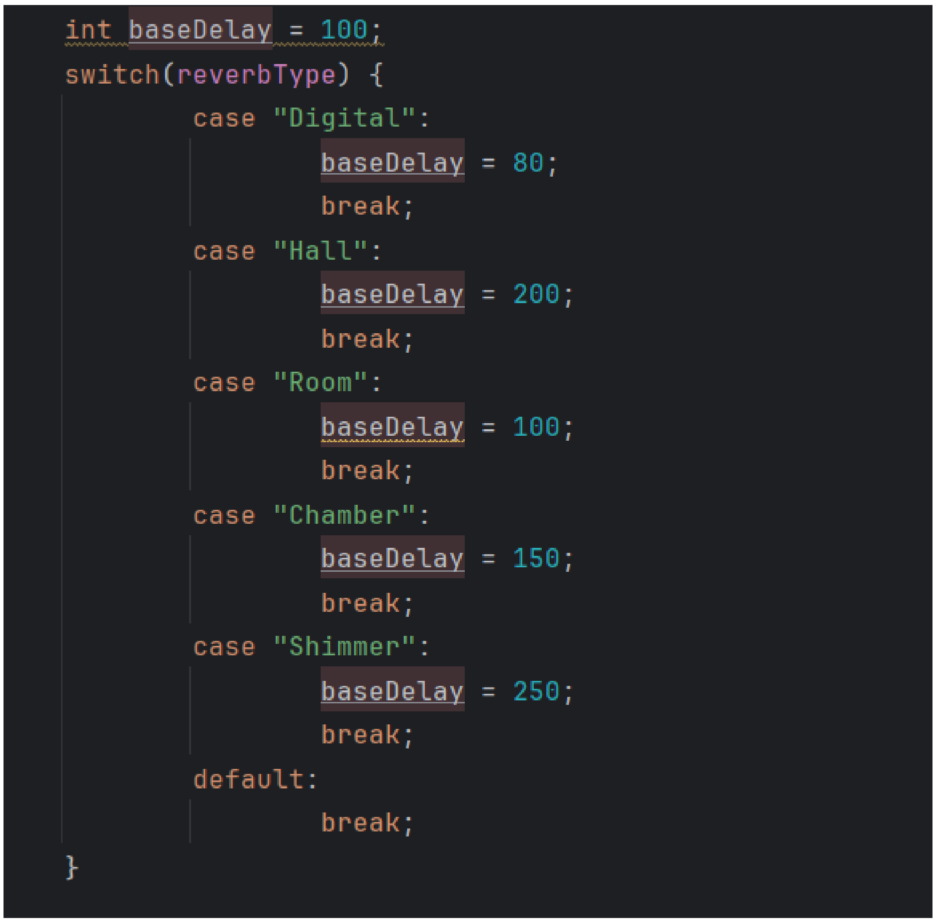
```
public void playNoteWithReverb(int reverbStrength, int reverbLength, String reverbType) {
        int delay = 100 * reverbLength;
        int echoAmount = reverbStrength / 20;

        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:



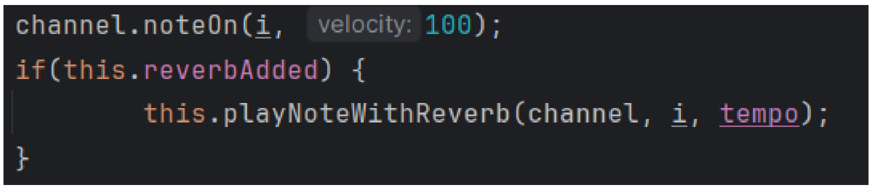
```
for(int i = 0; i < echoAmount; i++) {
        int echoVol = i * (reverbStrength / echoAmount);
        echoVol = 100 - echoVol;
        if(echoVol < 20){
                echoVol = 20;
        }

        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;
}
```

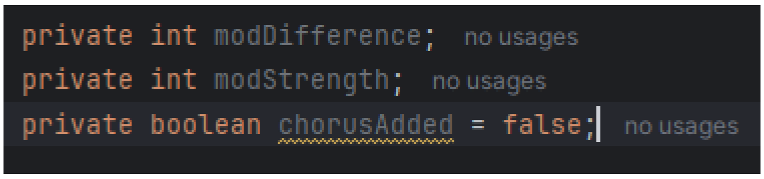
I then implemented this method into the play() method within the instrument class so that I could test it. This was done relatively simply, just by adding a way of checking if reverb is added to that instrument, and if it had you would run the method. This was done just after the note had been to reduce 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);
}
```

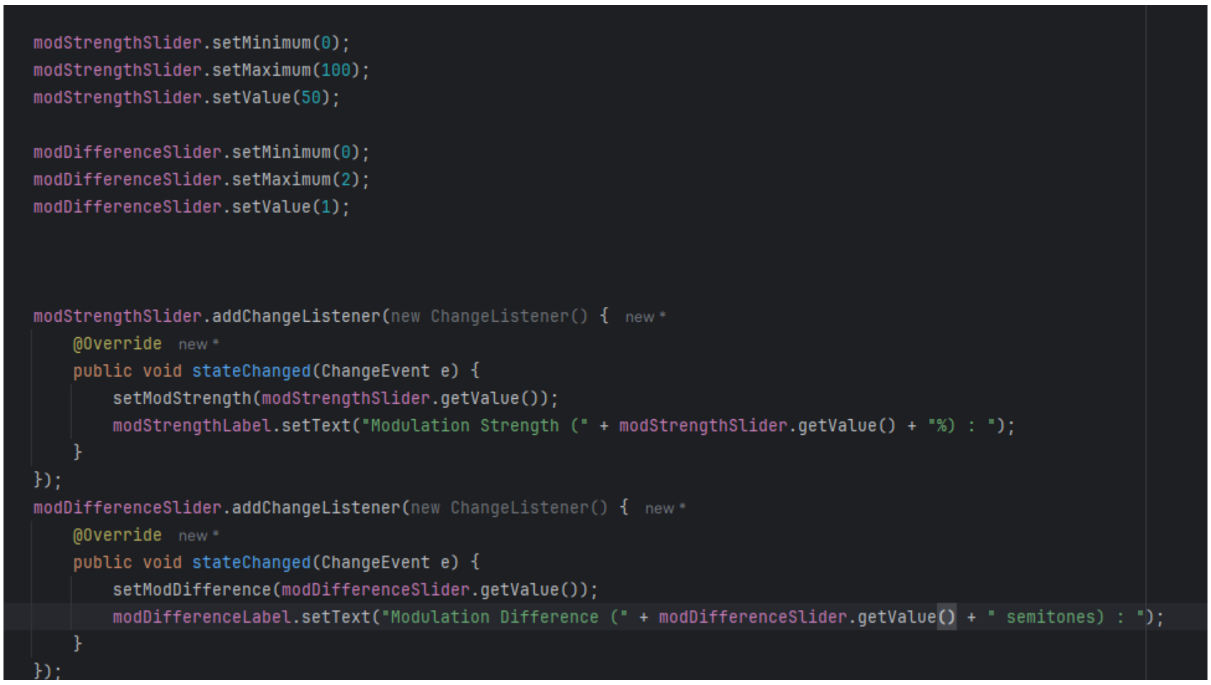
I then tested this implementation for different values of reverbLength, reverbStrength and reverbType and it worked as expected (i.e. the notes reverbed at different times/strengths/volumes depending on the values that I inputted after adding notes to an instrument.

The next step was to create a way of implementing chorus into a sequence. The first step for this was creating attributes within the instrument class to hold values for modDifference and modStrength. This will allow the user to change the type of chorus effect that is added into the sequence. This was relatively simple as I just looked at the data dictionary that I made prior to coding this section to find the data type and name of each attribute and then declare at the top of my instrument class. Here is that code:



```
private int modDifference;  no usages
private int modStrength;  no usages
private boolean chorusAdded = false;  no usages
```

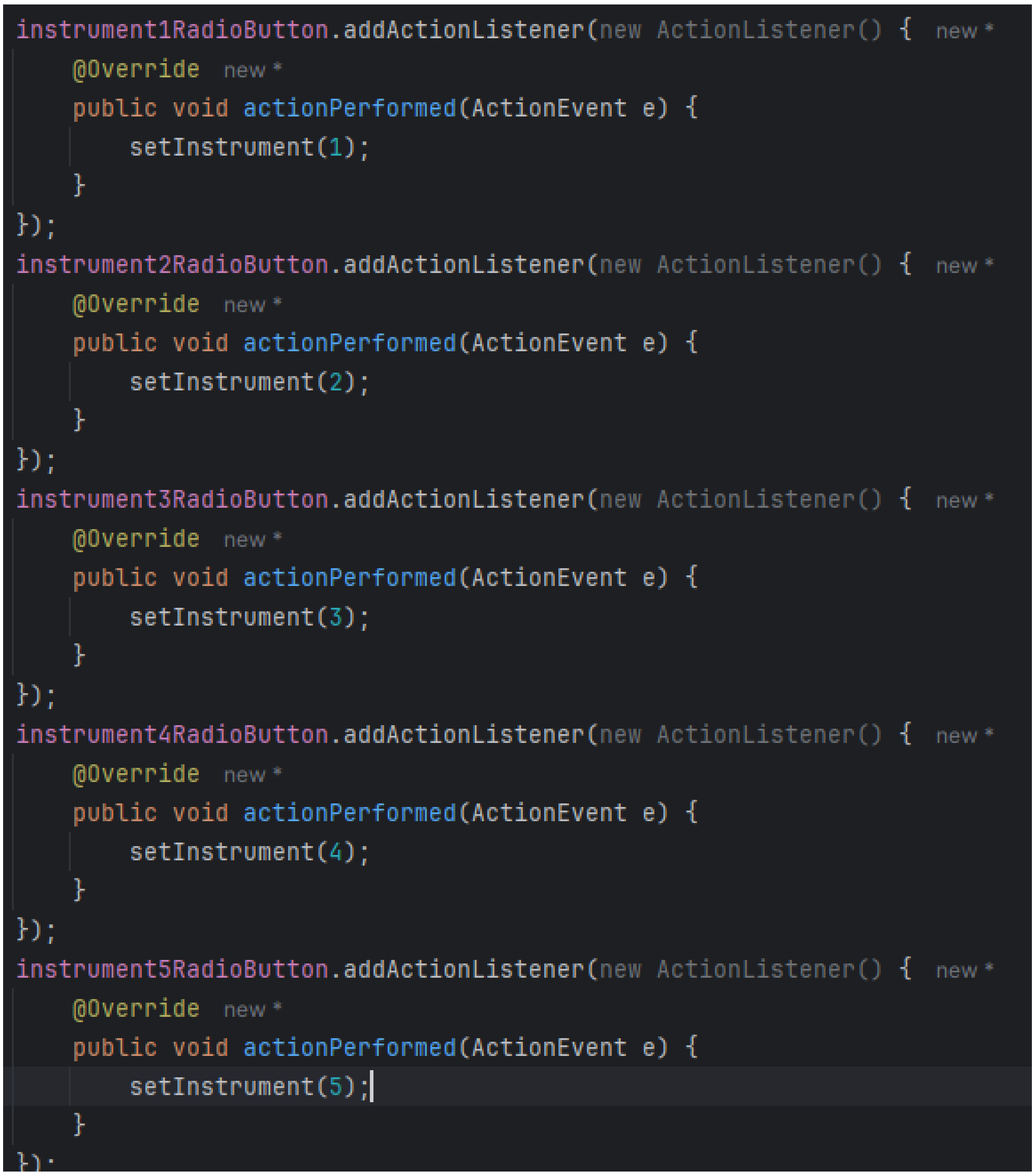
Next, I implemented change listeners for each slider within the chorusForm. This allows the user to manipulate their chorus effects in terms of strength and modulation difference. Within these change listeners I will call the respective setters for the modDifference and modStrength attributes within the chorus class. Also, I will change the modStrengthLabel and modDifferenceLabel to display the current values for the changing attributes. This allows the user to see what the slider is currently notating. This is the code for this module:



```
modStrengthSlider.setMinimum(0);
modStrengthSlider.setMaximum(100);
modStrengthSlider.setValue(50);

modDifferenceSlider.setMinimum(0);
modDifferenceSlider.setMaximum(2);
modDifferenceSlider.setValue(1);



modStrengthSlider.addChangeListener(new ChangeListener() {  new *
    @Override  new *
    public void stateChanged(ChangeEvent e) {
        setModStrength(modStrengthSlider.getValue());
        modStrengthLabel.setText("Modulation Strength (" + modStrengthSlider.getValue() + "%) : ");
    }
});
modDifferenceSlider.addChangeListener(new ChangeListener() {  new *
    @Override  new *
    public void stateChanged(ChangeEvent e) {
        setModDifference(modDifferenceSlider.getValue());
        modDifferenceLabel.setText("Modulation Difference (" + modDifferenceSlider.getValue() + " semitones) : ");
    }
});
```

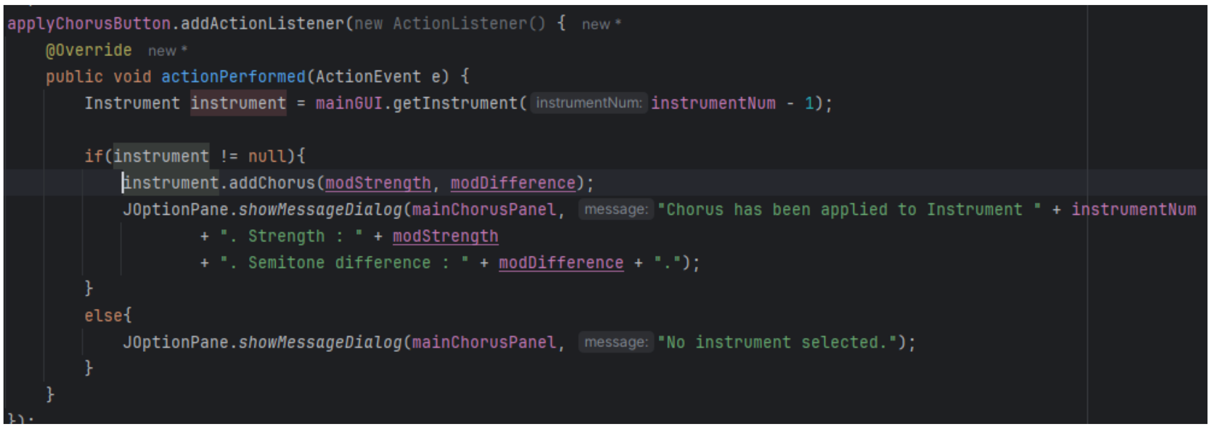
As you can see, I have also set the maximum and minimum values for each of these sliders too. Since the modStrength is a percentage, I set the maximum and minimum values as 0 – 100. Since the modDifference is measured in semitones, I set the maximum and minimum values as 0 – 2. The reason that this slider does not go high is because anything above 2 semitones would be too noticeable as a real note rather than chorus. I made sure that the user is aware of these units of measurements by adding them into the label.

The next step was to create an actionListener for each radioButton, so that I could set the instrumentNum to the correct number corresponding to the correct instrument. This means that chorus will be added to the correct instrument when implemented. This was done by simply creating five action listeners (one for each of the possible instruments) and inside them, calling the setInstrument method with the parameters 1 through 5. This code will correctly set chorus to be added to the correct instruments. Here is the code:



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

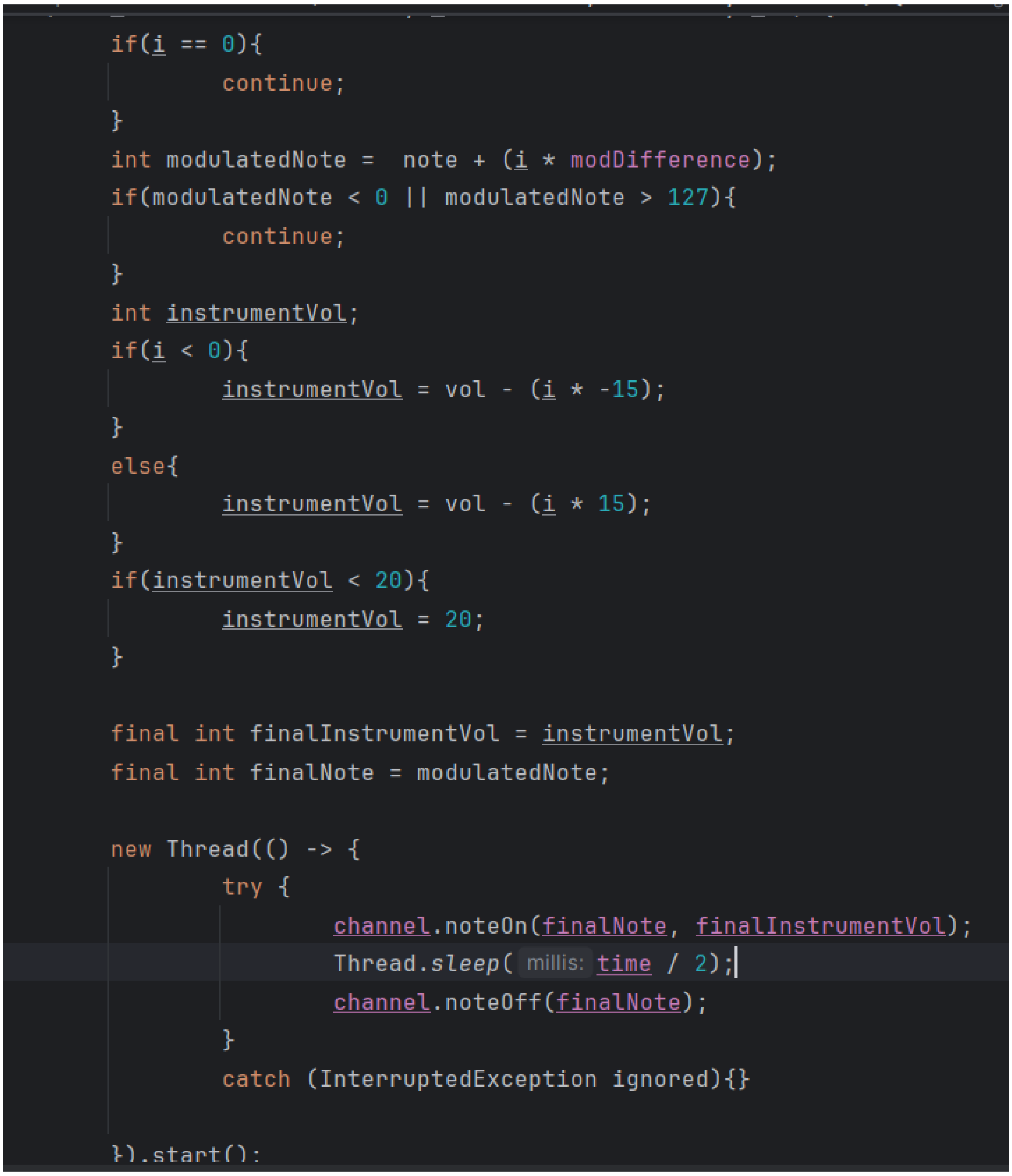
Next, I had to create a way of applying this chorus effect with the correct attributes to an instrument. This is done by creating an action listener for the applyChorusButton which holds code inside to create an Instrument attribute which holds the correct applying instrument, and if this is not null, call the addChorus method (not yet created) inside the instrument class. To make sure that the user is aware of the chorus being added, I will create an information message to show up. This message will show that chorus has been added and display the attributes for modDifference and modStrength. Here is the code for this module:



```java
applyChorusButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument(instrumentNum - 1);

        if(instrument != null){
            instrument.addChorus(modStrength, modDifference);
            JOptionPane.showMessageDialog(mainChorusPanel, "Chorus has been applied to Instrument " + instrumentNum
                    + ". Strength : " + modStrength
                    + ". Semitone difference : " + modDifference + ".");
        }
        else{
            JOptionPane.showMessageDialog(mainChorusPanel, "No instrument selected.");
        }
    }
});
```

I was not yet able to test this code since the addChorus method had not yet been created; however, I am sure that this will work by logic checking every statement.

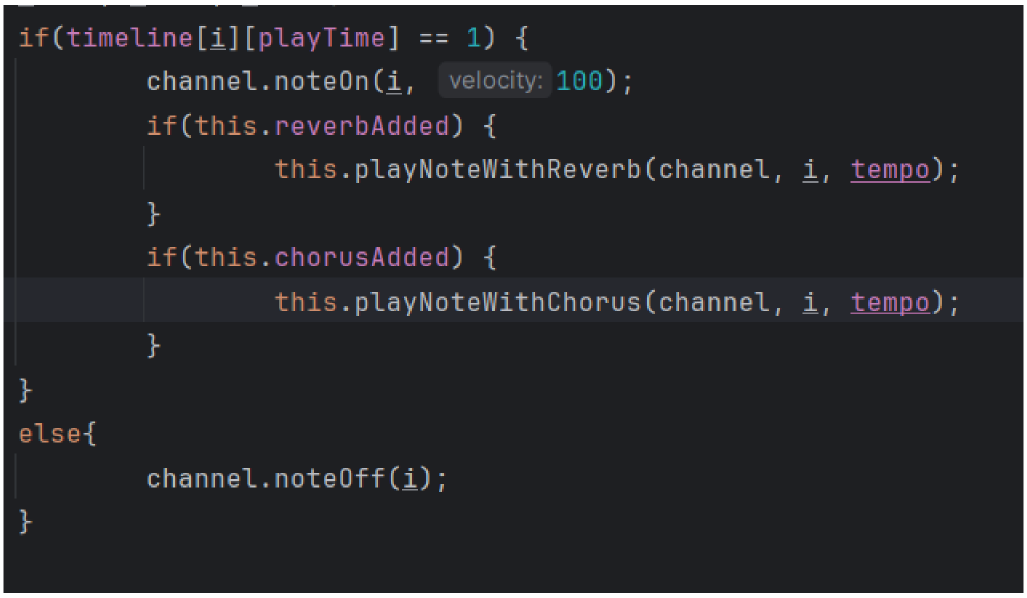
Next, I had to create a method called playNoteWithChorus() which was able to play a note, and the modulation of each note. This was done by first declaring the volume of the modulated notes, and the number of instruments that would be playing this note, based on the modStrength attribute. I then had to create a for loop to cycle through the possible amounts of voices and play different pitches and different volumes. When coding this section, I realised that I had to skip certain iterations of the loop (i.e. don’t play transposed notes that are outside of the midi range, and don’t play the original note). I did not yet know how to do this, so I referred to geeksforgeeks (Break and Continue statement in Java - GeeksforGeeks) to show me how to do this. It taught me that I must use a continue; statement. This is the code for this method:



```
        if(i == 0){
                continue;
        }
        int modulatedNote =  note + (i * modDifference);
        if(modulatedNote < 0 || modulatedNote > 127){
                continue;
        }
        int instrumentVol;
        if(i < 0){
                instrumentVol = vol - (i * -15);
        }
        else{
                instrumentVol = vol - (i * 15);
        }
        if(instrumentVol < 20){
                instrumentVol = 20;
        }

        final int finalInstrumentVol = instrumentVol;
        final int finalNote = modulatedNote;

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

        }).start();
```

As you can see, I have made slightly detuned copies of the chorus notes and played them at the same time as the original note. These notes will only play for half of the tempo so that they are not as noticeable as the original note. This will allow the user to create more full sounding music with Musify. I have also made it so that the instrumentVol will never go below 20, so that the modulated notes can always be heard. The instrumentVol also depends on the strength of the modulation, as shown within the if statements for negative and positive values for i.

The next step was to implement the playNoteWithChorus method, so that if chorus is added, the note will be played with that effect. This will allow the user to have more usability of my program, meaning that they can create more full and professional music. I will implement this method into the play() method, so that the note can be played at the same time as other modulated notes. Here is the code for that section:



```
if(timeline[i][playTime] == 1) {
        channel.noteOn(i, velocity: 100);
        if(this.reverbAdded) {
                this.playNoteWithReverb(channel, i, tempo);
        }
        if(this.chorusAdded) {
                this.playNoteWithChorus(channel, i, tempo);
        }
}
else{
        channel.noteOff(i);
}
```

This block of code simply checks if the chorusAdded attribute is true and if it is, the playNoteWithChorus() method is run with the correct parameters: channel (the midi channel), i (the note), and tempo (the tempo of the music). I tested this by adding notes into a sequence on an instrument. I then played it, added chorus, the played it again. I did notice that the chorus effect did sound slightly strange since it was playing notes semitones apart, whereas a regular chorus effect would slightly alter the frequency to avoid it sounding too "mashed together". However this is something that would take a significant amount of time for me to code, so I will stick to this being the effect for the moment. This was the last part of this stage, since I have 2 effects that work as good as I can implement them. This means that I could move onto testing and analysis.

Testing and Analysis:

Test numbe r	Description of test	Test data	Expected outcome	Achieved?
1	Test adding notes, playing them, adding reverb, then playing them again	Input notes into individual instrument, play, add reverb, play again	Notes should play with reverb after it has been added	YES
2	Test adding different types of reverb	Input notes into individual	The different types of	YES

		instrument, add one reverb type, play, add another reverb type, play, repeat	reverb should have different delays between echoes	
3	Test adding different strengths of reverb	Input notes into individual instrument, play, add reverb of one strength, play, add different reverbStrengt h value, play again, repeat	The volume and quantity of echoes should increase with reverb strength	YES
4	Test adding different length of reverb	Input notes into individual instrument, play, add reverb of one length, play, add different reverbLength value, play again, repeat	The length of reverb should increase with increased reverbLength values	YES
5	Test adding reverb to multiple instruments	Input notes into multiple instruments, add reverb to multiple instruments, play everything separately AND together	Reverb effect should be heard when playing it all together, and individually	YES
6	Test adding chorus	Input notes, play, add chorus effect, play again	Chorus effects should be heard during second playback	SOMEWHAT ->explained in analysis
7	Test adding different strength of chorus	Input notes into individual instrument, play, add chorus of one	The volume of modulated notes should increase with	YES

		strength, play again, add different value of chorusStrength, play again	chorus strength	
8	Test adding different modulation differences of chorus	Input notes into individual instrument, play, add chorus of one modulation difference, play, change value for chorus modulation difference, play again	The number of modulated notes should increase in semitone difference against the original note with the modulation difference	YES
9	Test adding chorus to multiple instruments	Input notes for multiple instruments, play, add chorus to all instruments, play separately AND together	The chorus effects should be hear when playing multiple instruments together and separately	SOMEWHAT ->explained in analysis

Analysis:

During my testing of this stage, I noticed that my chorus effect had some problems. This problem was that it did not sound how chorus effect normally sounds. I did some research and found out the reason for this is that the difference between modulated notes and the original notes is measured in fractions of frequency, however in my code, I was measuring it in semitones (A LOT larger of a difference). This makes my chorus effect sound a lot more broken up and distorted. I do not however have time to fix this problem, since coding in a way of changing the frequency of notes rather than the pitch (in semitones) is very complex and time consuming. For this reason, I will leave my chorus effect how it is.

On the other hand, I was very happy with how my reverb effect sounded. The echoes came out very well, and the difference between different types of reverbs was clear in the sound. This is good because it will allow the user to change their music in a lot more ways, meaning that my program is more usable and suits my user more for their needs.

Testing my effects with different values was also interesting, because I noticed that when the reverbStrength or modStrength goes below a certain level, the values start to have no difference on the effect. This is because within the code for playNoteWithReverb and playNoteWithChorus methods, I have made it so that when these attributes go below a certain value, they are set back up to a base value. This was done so that the effect is still heard. Therefore, I do not see this as a problem, because without it, the implementation of the effects would be unclear. This means that my program will not mislead the user into misusing the effects that they have available, overall increasing the usability and functionality of my program.

Stage 3 – Advanced Sequencing features:

- In this section, I will be implementing various useful features for the user. One of these is being able to resize notes, which is essential in my program, since it allows the user to add different lengths of sounds. This is a feature that is common in many other music sequencing software that I have researched. I will implement this feature to make sure my program has more functionality and is able to adapt to what the user wishes to do.

- Also, I will add a feature to allow zooming in and out on the timeline. This feature will be useful since it will allow the user to view all parts of the timeline as they wish. In addition, this feature will mean that the user can focus into the parts of the sequence that they are working on. This increases usability of my program. Furthermore, allowing zoom on my sequencer will make my program more accessible, since those who have some visual impairments will be able to see my timeline on a much larger scale (when zoomed in). This makes my program available to a wider audience.

- Another feature that I plan on implementing in this section is the ability to remove effects from certain instruments. I noticed when testing the final section that to remove effects I had to stop and rerun the program, which is not ideal. This feature will make my program easier to test whilst also making it more functional for the user. Overall, this feature will make my program more user-friendly.

- By the end of this section, my user should have more available features to make sequencing music more accessible and advanced, allowing them to make more complex pieces.

Design:

Algorithm design:

- For note resizing, I will create a new class, called note. This class will be used as a list in the instrument class instead of the current timeline array. This note class will be relatively simple, however will allow me to hold not only when the note starts, but also when the note ends. This means that I will be able to make different lengths of notes, which is the aim of part of this section. To make this note class work with other classes, I will have to make small alterations to the instrument and sequencerGrid classes to work with this new implementation of a timeline. Here is the pseudocode for the note class, and the changes that I will make in the instrument and sequencerGrid classes:

```
Public class Note {

    Private int pitch;

    Private int start;

    Private int length;


    Public Note(int pitch, int length, int start){

        This.start = start;

        This.pitch = pitch;

        This.length = length;

    }

    Public int getPitch(){

        Return pitch;

    }

    Public int getLength(){

        Return length;
```

```
}
Public int getStart(){
        Return start;
}
Public void setPitch(int pitch){
        This.pitch = pitch;
}
Public void setStart(int start){
        This.start = start;
}
Public void setLength(int length){
        This.length = length;
}
Public Boolean containsCell(int col){
        IF (col >= start) && (col < (start + length)) THEN
                Return true;
        ELSE
                Return false;
        ENDIF
}
Public Boolean containsPoint(int x, int y, int cellSize){
        Int left = start *cellSize;
        Int right = (start + length) * cellSize;
        IF (x >= left) && (x < right) THEN
                Return true;
        ELSE
                Return false;
        ENDIF
```

```
    }
```

- The next code that I will write will be for the instrument class. I will not include anything that has not been changed and explain in comments if I have replaced anything. This will make it easier for me to refer to when I start developing this in my java project. Here is the added and changed pseudocode for the instrument class:

```
//replacing timeline 2D array
Private List<Note> notes = new ArrayList<>();


//new note model
Public void addNote(Note note) {
    This.notes.add(note);
}
Public void removeNote(Note note) {
    This.notes.remove(note);
}
Public list<Note> getNotes() {
    Return notes;
}


//within play method changes in for loop to play notes
For(int I = 0; I < notes.size(); i++){
    IF  note.start == playtime THEN
            Channel.noteOn(note.pitch, 100);
    IF note.start + note.length == playtime THEN
            Channel.noteOff(note.pitch);
}
```

- These changes will make my new note class work alongside the instrument class, resulting in notes being able to have duration. The next pseudocode will show how the sequencerGrid will change to make the user able to drag the end of the notes to resize them. Also, other simpler changes are made to make sure that the new note class works with the code within this class. In this pseudocode block I will only show new or changed parts and annotate these with comments to make the implementation during the development process less complex. Here is the pseudocode:

```
Private Note selectedNote = null;

Private Boolean resizing = false;


Private Note findNoteAt(int pitch, int col){

    FOR Note note IN instrument.getNotes(){

        IF notePitch = pitch && col >= note.start && col < note.start +
note.length THEN

            Return note;

        ENDIF

    }

    Return null;

}


//new code within mouseListener for mousePressed


selectedNote = findNoteAt(pitch, col)


IF selectedNote != null && col = selectedNote.start + selectedNote.length THEN

    Resizing = true;

ELSE IF selectedNote == null THEN

    Note newNote = newNote(pitch, col, 1);
```

```
        Instrument.addNote(newNote);
        selectedNote = newNote;
ENDIF


//new code within mouseListener for mouseReleased
        {
                selectedNote = null;
                resizing = false;
        }
//new code within mouseListener for mouseDragged


IF selectedNote != null && resizing = true THEN
        Int col = e.getX() / cellSize;
        selectedNote.length = col – selectedNote.start;
ENDIF


//new code within paintComponent to draw rectangles of notes for extended
lengths, rather than single cells


FOR note IN instrument.getNotes() {
        Int row = 127 – note.pitch;
        Int x = note.start * cellSize;
        Int y = row * cellSize;

        setColor(black);
        fillRectangle(x, y, note.length * cellSize, cellSize);
}
```

These updates in these classes should make my sequencer work with the new model for the timeline as an arraylist of the note class. This means that I am able to add duration to each note which will give more functionality and usability to my user.

The next block of code that I will show is the code for removing effects. This is a relatively simple module, since it only involves creating a button in the effects page to remove the effect. This will be done in either the reverbEffectPanel or chorusEffectPanel based on which effect they would like to remove. This button will have an action listener which removes the effect from the desired instrument. I will also add a group of radio buttons so that the desired instrument can be selected. Additionally, a simple removeReverb method will be created in the instrument class to complete the desired action of removing reverb or chorus. This will simply set reverb or chorus to null. Here is the pseudocode for this module:

```
reverbRemoveButton.addActionListener(ActionEvent e){
    switch(removeInstrumentRadioGroup.getValue){
        case “Instrument 1”:
            mainGUI.getInstrument(0).removeReverb();
            break;
        case “Instrument 2”:
            mainGUI.getInstrument(1).removeReverb();
            break;
        case “Instrument 3”:
            mainGUI.getInstrument(3).removeReverb();
            break;
        case “Instrument 4”:
            mainGUI.getInstrument(4).removeReverb();
            break;
        case “Instrument 5”:
            mainGUI.getInstrument(5).removeReverb();
            break;
        default:
```

```
                        break;
                }
        }
```

Above is the code that will be implemented into the reverb class.

Below is the code that will be implemented into the chorus class:

```
        chorusButton.addActionListener(ActionEvent e){
        switch(removeInstrumentRadioGroup.getValue){
                case “Instrument 1”:
                        mainGUI.getInstrument(0).removeChorus();
                        break;
                case “Instrument 2”:
                        mainGUI.getInstrument(1).removeChorus();
                        break;
                case “Instrument 3”:
                        mainGUI.getInstrument(3).removeChorus();
                        break;
                case “Instrument 4”:
                        mainGUI.getInstrument(4).removeChorus();
                        break;
                case “Instrument 5”:
                        mainGUI.getInstrument(5).removeChorus();
                        break;
                default:
                        break;
                }
        }
```

As you can see the code for removing the chorus and reverb effect is similar. This is because the only purpose of this listener is to set the instrument that the effect is being removed from and then run a method in the instrument class to do the actual action.

Next, I will show you the code that I will implement into the instrument class for removing the effect. This is very simple as it is simply setting either the chorus or reverb attribute to null. Here is the code for this section:

```
Public procedure removeReverb(){
    This.reverb = null;
}
Public procedure removeChorus(){
    This.chorus = null;
}
```

From prior knowledge, I believe that there is a way to add zooming into the sequence via IntelliJ idea without necessary code. Therefore I have not included the pseudocode for this.

Data:

- Below I will create a data dictionary to hold all the attributes and methods that are necessary for development in this stage. This will make it easier during development, since it will allow me to refer back to it to better understand what my attributes and methods purposes are and the type that they return

Attributes:

Name	Type	Description	Held in class:	Local/Global
Pitch	Int	The midi pitch of a note	Note	Global
Start	Int	Start time of this note in a grid	Note	Global

Length	Int	Length of a note in a grid	Note	Global
Notes	List<notes>	Replacement of timeline array, and holds all notes on the timeline	Instrument	Global
selectedNote	Note	The current note being manipulate	seqeuncerGrid	Global
Resizing	Boolean	True when a note is being manipulate	sequencerGrid	Global

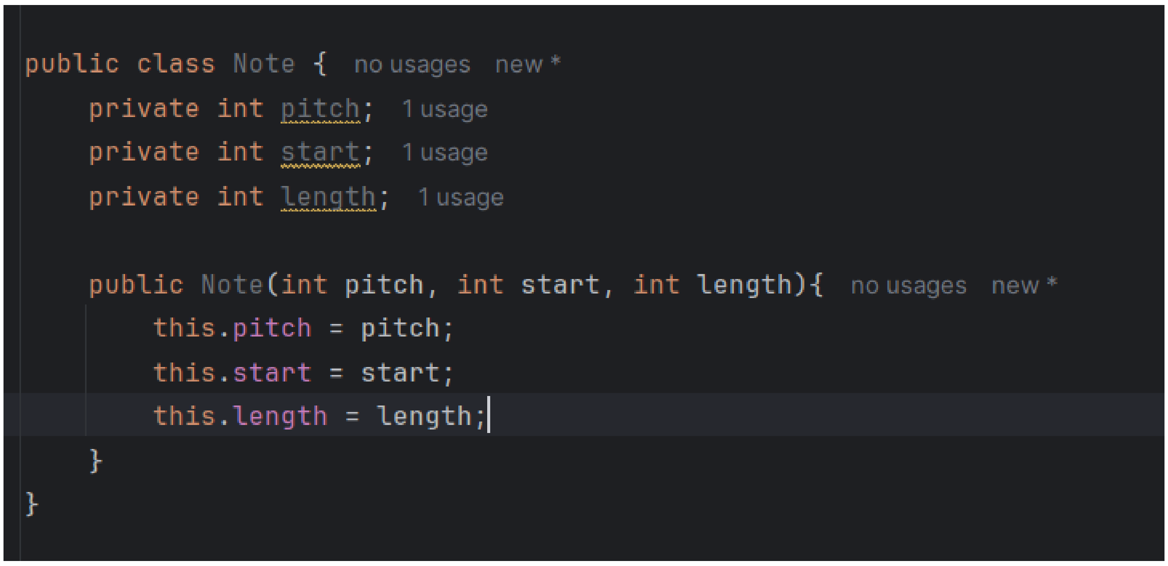
Methods:

Name	**Return type**	**Description**	**Held in class:**
getPitch()	Int	Getter for pitch attribute	Note
getStart()	Int	Getter for start attribute	Note
getLength()	Int	Getter for length attribute	Note
setPitch(int pitch)	Void	Setter for pitch attribute	Note
setStart(int start)	Void	Setter for start attribute	Note
setLength(int length)	Void	Setter for length attribute	Note
addNote(Note note)	Void	Adds a note object to the notes array list (timeline)	Instrument
removeNote(int pitch, int time)	Void	Removes a note object from the	Instrument

		notes array list (timeline)	
getNotes()	List<notes>	returns the list of notes	Instrument
findNoteAt(int pitch, int col)	Note	Finds a specific note at the desired pitch and place	sequencerGrid

Development:

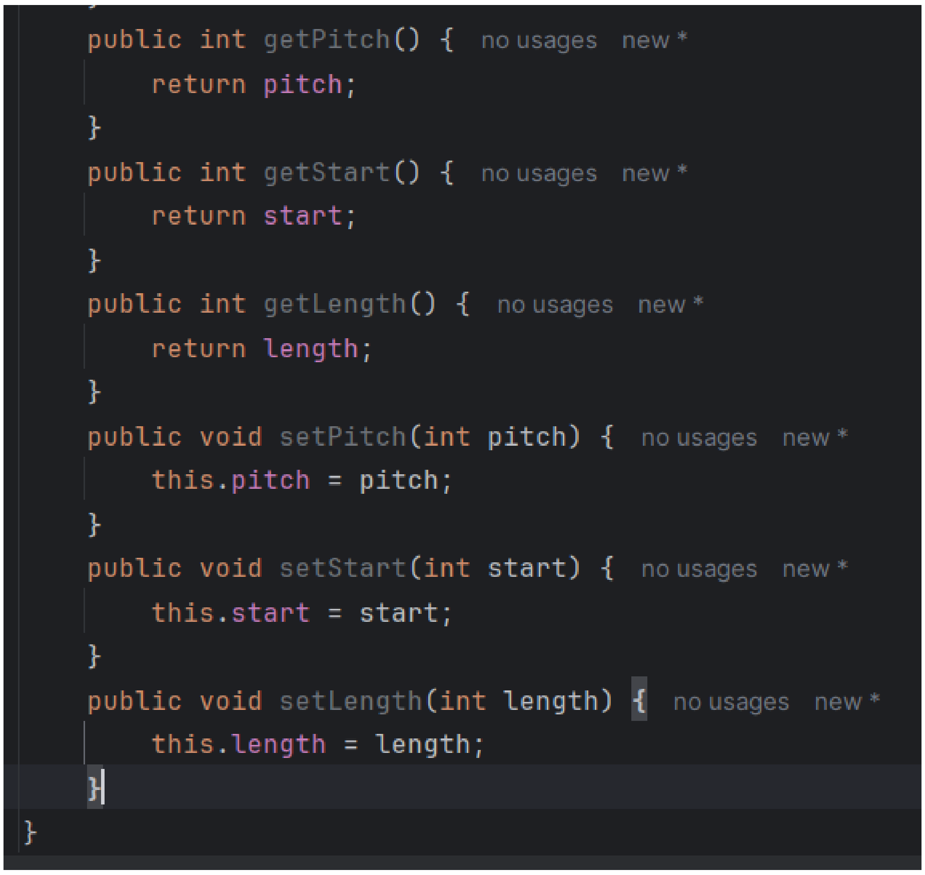
For developing this stage, I decided to start by creating the note class. This is because a lot of the other coding in this section relies on this class working properly. For example, the altered methods in the instrument and seqeuncerGrid class will not be able to work without this one working properly. To start with, I declared all the attributes and the constructor, since these are the backbone of all classes. Here is that code:



```
public class Note {  no usages  new *
    private int pitch;  1 usage
    private int start;  1 usage
    private int length;  1 usage

    public Note(int pitch, int start, int length){  no usages  new *
        this.pitch = pitch;
        this.start = start;
        this.length = length;
    }
}
```

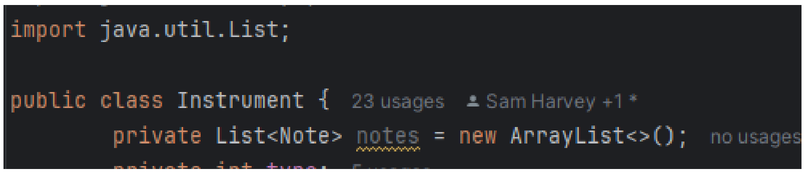
This is simple code which just declares all the attributes for this class within the constructor. I did this so that when a note is created, all relevant information about it must be inputted so that the note can be played and manipulated correctly.

Next, I created all the getters and setters for these attributes. This was done to ensure that my code is fully encapsulated, meaning that my attributes cannot be manipulated incorrectly. This was very simple to do as I referred to the pseudocode that I made previously. Here is the code for this module:

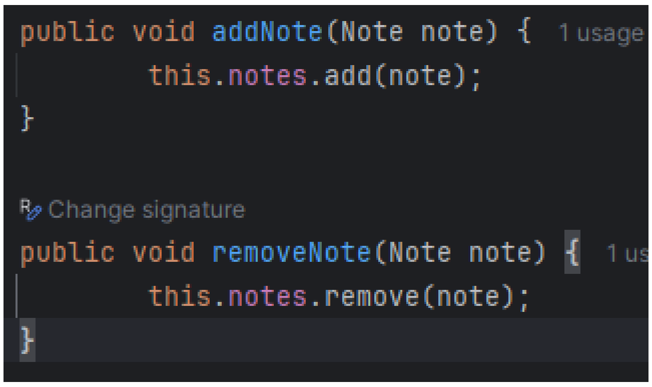

```
    public int getPitch() {  no usages   new *
        return pitch;
    }
    public int getStart() {  no usages   new *
        return start;
    }
    public int getLength() {  no usages   new *
        return length;
    }
    public void setPitch(int pitch) {  no usages   new *
        this.pitch = pitch;
    }
    public void setStart(int start) {  no usages   new *
        this.start = start;
    }
    public void setLength(int length) {  no usages   new *
        this.length = length;
    }
}
```

This code ensures that the note class is fully encapsulated, and therefore increases the reliability of my program, since I am unable to accidentally and incorrectly change these attributes from another class.

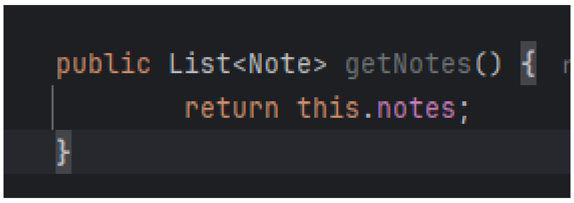
The next part that I coded was changing the timeLine attribute in the instrument class to a list of note objects. I first started by declaring this variable in the way described in my pseudocode. This will allow me to change the length of notes much easier. Here is the code for that section:


```
import java.util.List;

public class Instrument {  23 usages   Sam Harvey +1 *
        private List<Note> notes = new ArrayList<>();  no usages
```

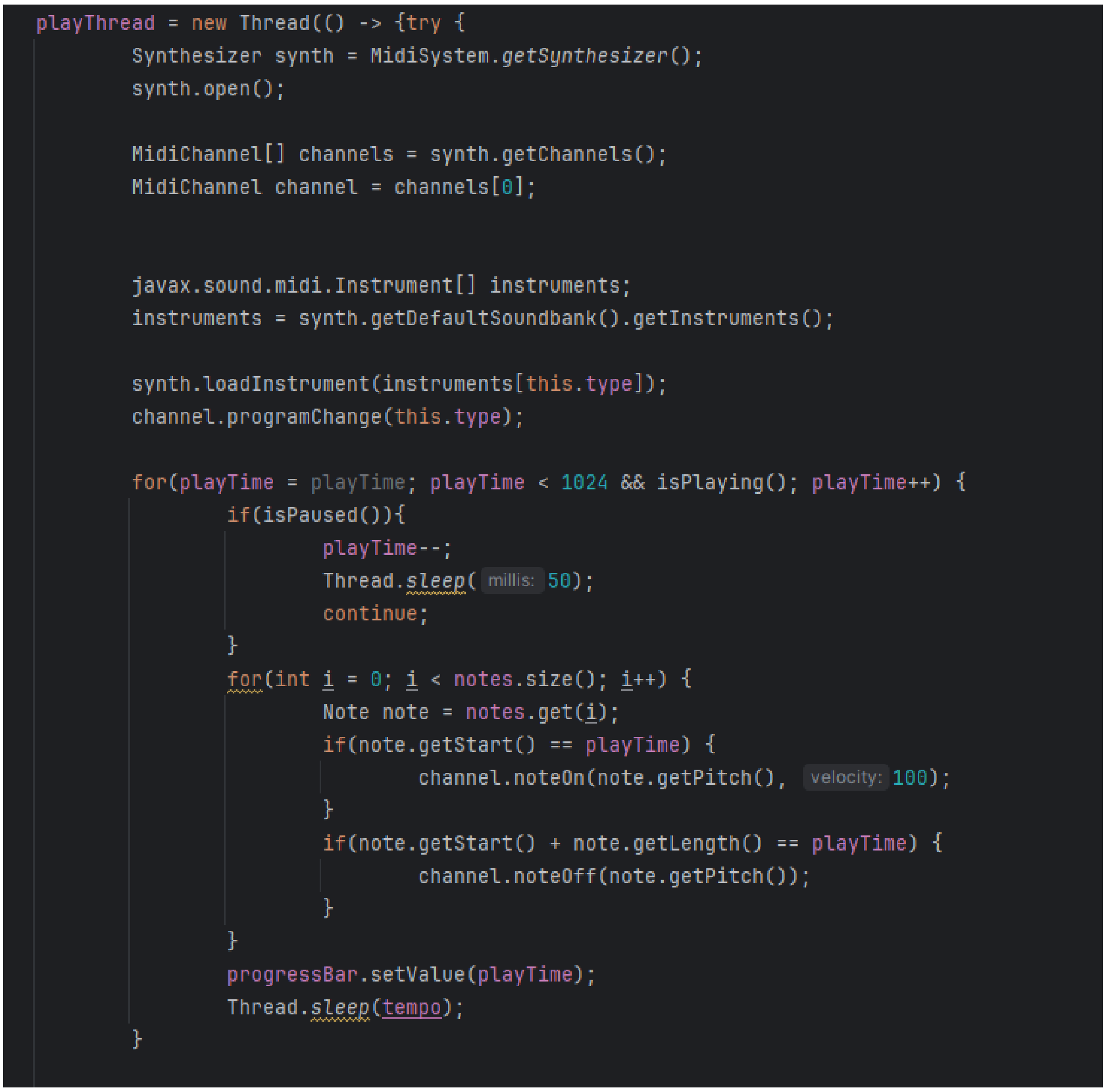
As you can see, I also had to import java.util.List to make this code work. The next step was to implement this new list in all the places where timeLine was previously used. I did this by looking through each class, starting with instrument, and looking where errors occurred. Fixing these syntax errors will mean that my code will be able to run as intended. I will paste all the small changes that I make here:



```
public void addNote(Note note) {  1 usage
        this.notes.add(note);
}

Change signature
public void removeNote(Note note) {  1 us
        this.notes.remove(note);
}
```

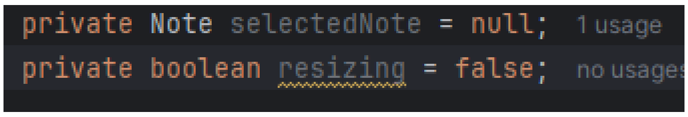


```
public List<Note> getNotes() {  r
        return this.notes;
}
```

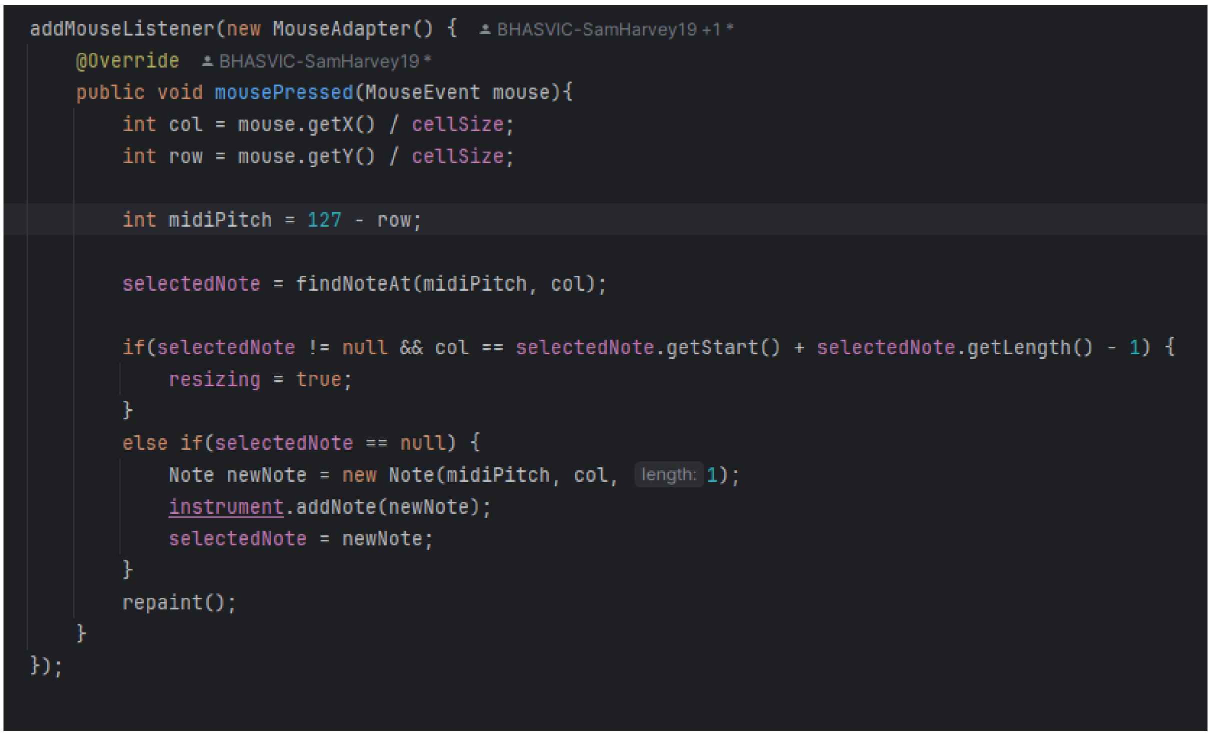


```
playThread = new Thread(() -> {try {
        Synthesizer synth = MidiSystem.getSynthesizer();
        synth.open();

        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 < 1024 && isPlaying(); playTime++) {
                if(isPaused()){
                        playTime--;
                        Thread.sleep( millis: 50);
                        continue;
                }
                for(int i = 0; i < notes.size(); i++) {
                        Note note = notes.get(i);
                        if(note.getStart() == playTime) {
                                channel.noteOn(note.getPitch(),  velocity: 100);
                        }
                        if(note.getStart() + note.getLength() == playTime) {
                                channel.noteOff(note.getPitch());
                        }
                }
                progressBar.setValue(playTime);
                Thread.sleep(tempo);
        }
```

This is all the changed code in the instrument class. As you can see in the play() method, I have not yet made a way to play a note with the effect if they have been added. I will implement this after I have changed all the other classes to work with the new notes list. The next class that I changed is the sequencerGrid so that it was able to work with the notes list. Whilst doing this code, I also implemented the resizing note attributes so that this feature can be implemented later during coding. Here is all the changed code:



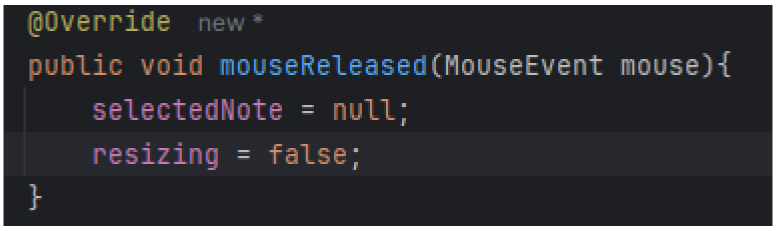
```
private Note selectedNote = null;  1 usage
private boolean resizing = false;  no usages
```



```
addMouseListener(new MouseAdapter() {  BHASVIC-SamHarvey19 +1 *
    @Override  BHASVIC-SamHarvey19 *
    public void mousePressed(MouseEvent mouse){
        int col = mouse.getX() / cellSize;
        int row = mouse.getY() / cellSize;

        int midiPitch = 127 - row;

        selectedNote = findNoteAt(midiPitch, col);

        if(selectedNote != null && col == selectedNote.getStart() + selectedNote.getLength() - 1) {
            resizing = true;
        }
        else if(selectedNote == null) {
            Note newNote = new Note(midiPitch, col,  length: 1);
            instrument.addNote(newNote);
            selectedNote = newNote;
        }
        repaint();
    }
});
```

The last pasted image shows the new mouseListener for mousePressed. This listener is used to detect when a user wants to add a note. The first if statement checks if the user has pressed the edge of a note to resize it, and if so, sets the resizing attribute to true. If it has not been resized, then a new note is added. This is only done if the note within the selected cell is null (doesn't yet exist). This code is essential for the user being able to both add notes and resize them.

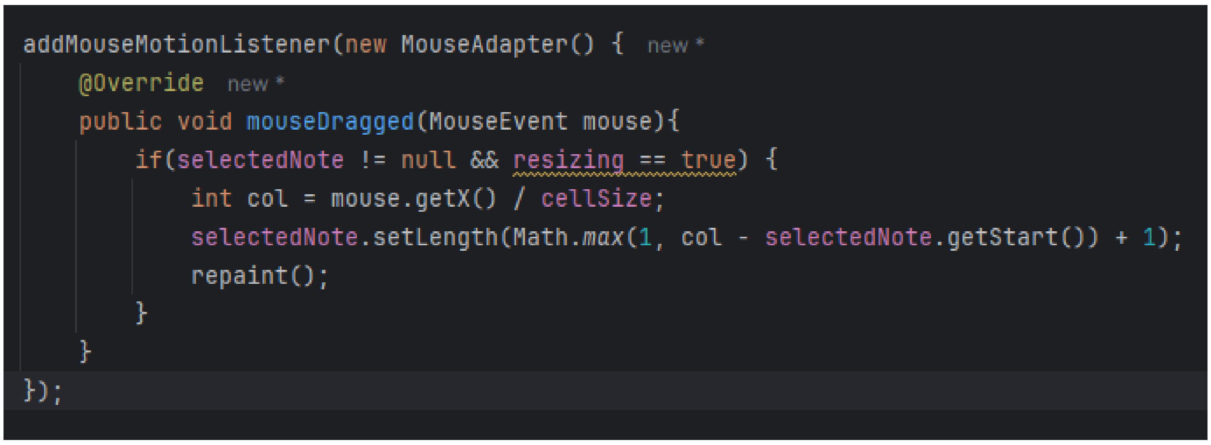
Next, I added another mouse listener for when the mouse is released. This was a relatively simple once and just involved setting the resizing attribute to false – because the user is no longer holding a note – and setting the selectedNote to null – because the user is no longer selecting the note. This is a simple yet useful module of code because it allows the code to know when to stop resizing a note. Here is that block of code:



```java
@Override  new *
public void mouseReleased(MouseEvent mouse){
    selectedNote = null;
    resizing = false;
}
```

The next stage was to add a mouseMotionListener, which inside has a mouseDragged method. This is so that we can detect if the mouse is being dragged, and if so, check if the selectedNote is not null and if the resizing attribute is true. Inside that if statement is a block of code that will resize the note based on the user's mouse movements. This module is the main code for resizing a note, since it holds the code to change the notes length. This is useful as it will allow my user to make more full and deeper sounding sequences.

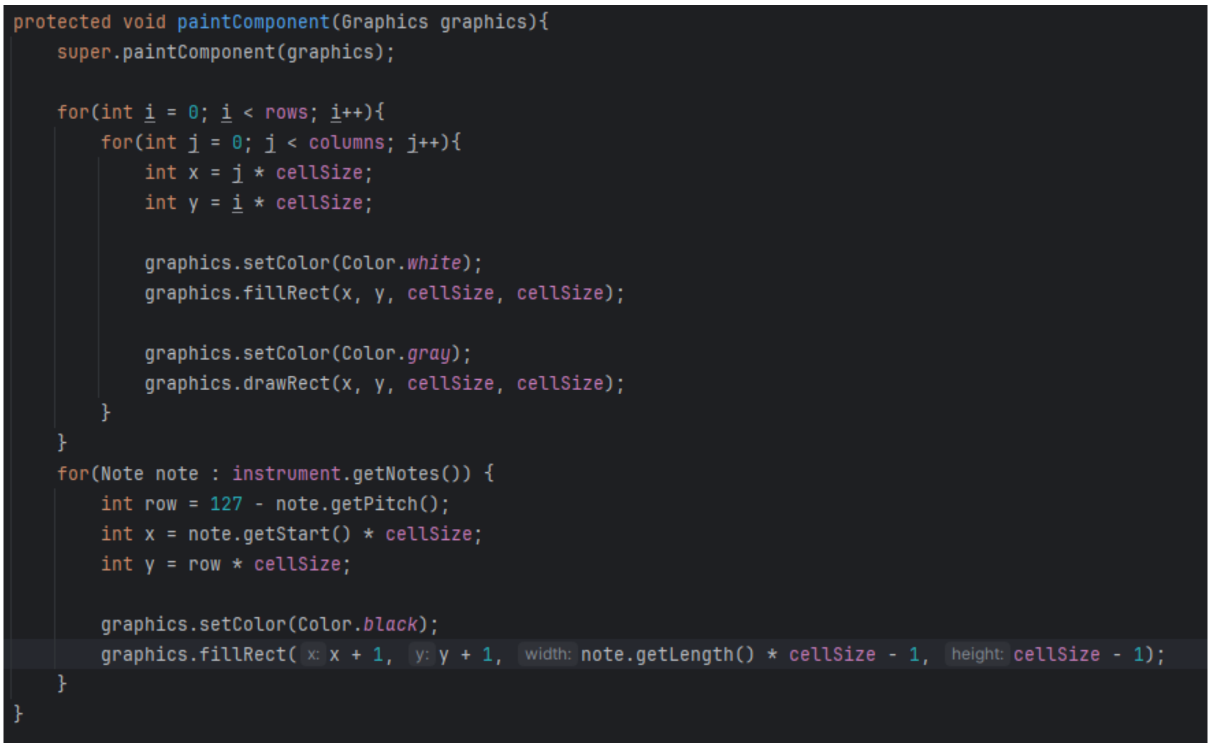
During the implementation of this listener, I realised that I needed to set the notes length to an integer. This was hard as the note can be dragged to a decimal value. To find out how to do this is visited geeksforgeeks (https://www.geeksforgeeks.org/java/java-math-max-method/) which taught me all that I needed to know. Here is the code for that section:



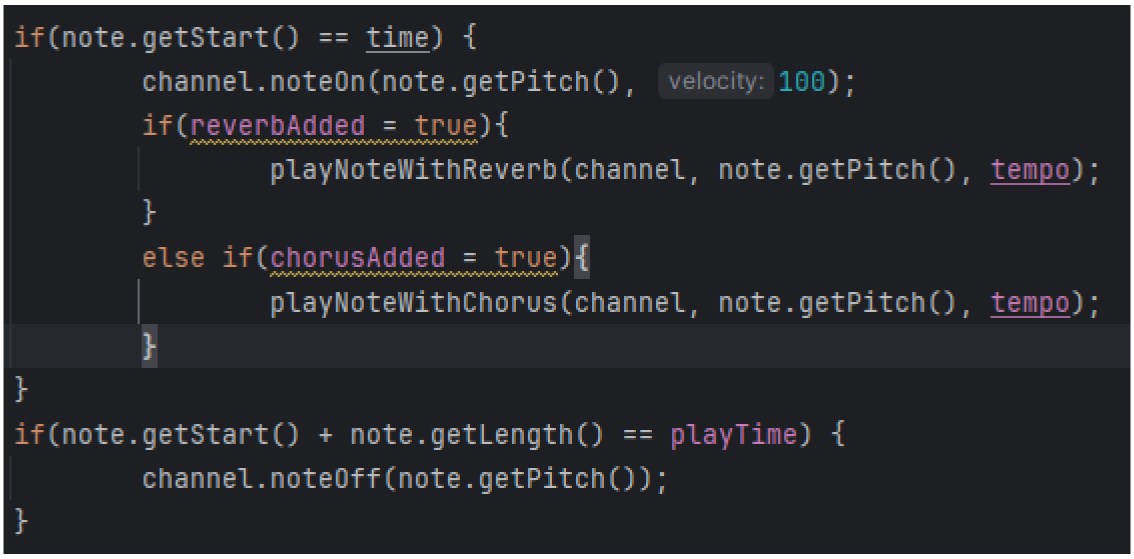
```java
addMouseMotionListener(new MouseAdapter() {  new *
    @Override  new *
    public void mouseDragged(MouseEvent mouse){
        if(selectedNote != null && resizing == true) {
            int col = mouse.getX() / cellSize;
            selectedNote.setLength(Math.max(1, col - selectedNote.getStart()) + 1);
            repaint();
        }
    }
});
```

This code will set "col" to a specific cell in the grid and then set the length of the selectedNote to a rounded number in relation to where the mouse has been dragged to. This means that the notes length can be changed, giving the user more functionality of my program. I was not yet able to test this code as other syntax errors were still present in my code, however the next step was to fix these errors so that I am able to test my code soon.

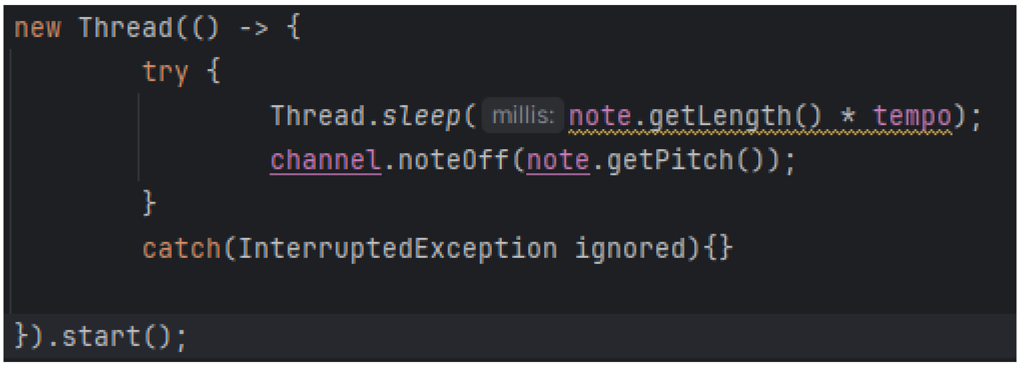
After that, I changed the paintComponent method to work with the new notes list. This involved running through all notes in the list, and then drawing a rectangle (block) related to the note and the length of the note. This means that the user to able to see what the length of the note is and is therefore able to use my program better. This gives more usability and functionality to my program. Here is the block of code:


```java
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.white);
            graphics.fillRect(x, y, cellSize, cellSize);

            graphics.setColor(Color.gray);
            graphics.drawRect(x, y, cellSize, cellSize);
        }
    }
    for(Note note : instrument.getNotes()) {
        int row = 127 - note.getPitch();
        int x = note.getStart() * cellSize;
        int y = row * cellSize;

        graphics.setColor(Color.black);
        graphics.fillRect( x: x + 1,  y: y + 1,  width: note.getLength() * cellSize - 1,  height: cellSize - 1);
    }
}
```

Now that all the code has been changed to work with the new notes list, I was able to go back to the play method in the instrument class and implement the effects. This was very simple to do, since I just had to check if reverbAdded and/or chorusAdded attributes were true, and if they were, run the playNoteWithReverb and/or the playNoteWithChorus methods. This means that if the user has added an effect, it will change how the sequence sounds during playback, giving more functionality and usability to my software. Here is that block of code:


```java
if(note.getStart() == time) {
        channel.noteOn(note.getPitch(),  velocity: 100);
        if(reverbAdded = true){
                playNoteWithReverb(channel, note.getPitch(), tempo);
        }
        else if(chorusAdded = true){
                playNoteWithChorus(channel, note.getPitch(), tempo);
        }
}
if(note.getStart() + note.getLength() == playTime) {
        channel.noteOff(note.getPitch());
}
```

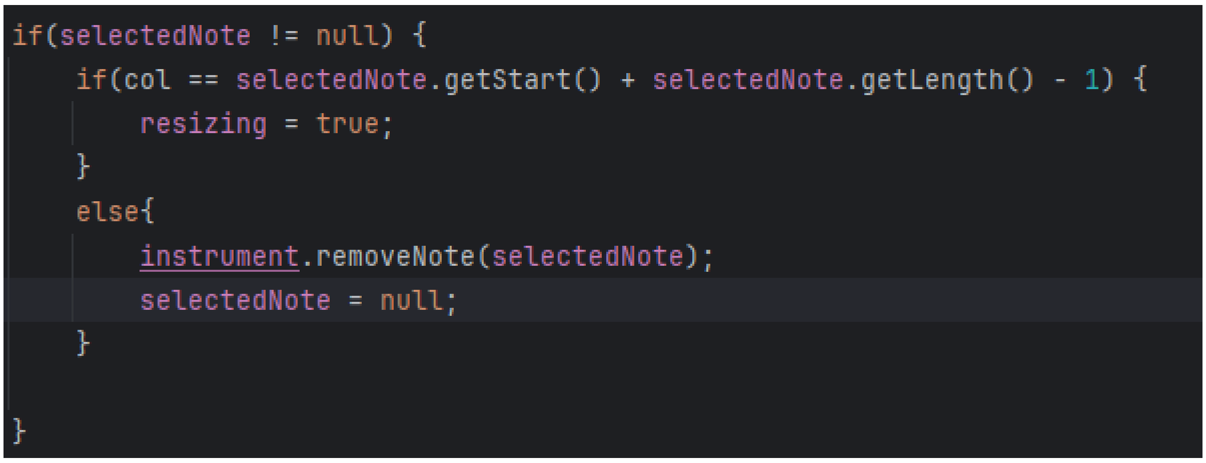
I then tested this code by adding notes, playing them, changing their duration by dragging them, and then playing them again to see if the note's duration changed. I found an error with this – that the notes duration change was inconsistent. To try and fix this, I made a thread within my play method which waits for (notes length * tempo) before changing the notes state back to off. Here is that block of code:



```
new Thread(() -> {
        try {
                Thread.sleep( millis: note.getLength() * tempo);
                channel.noteOff(note.getPitch());
        }
        catch(InterruptedException ignored){}

}).start();
```

When testing this, I was happy with the results, since the duration of each note seemed to be consistent with what I had inputted as the notes duration. This means that my user will be able to correctly change the duration of notes in their sequence, giving them more functionality and usability.

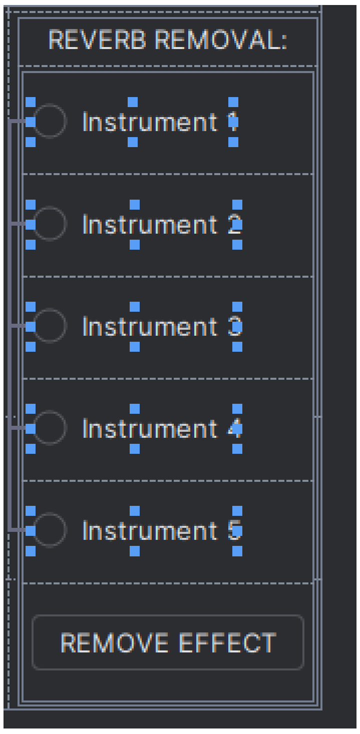
On the other hand, I noticed that there was no way to remove a note from the sequence, since when I clicked on a note to remove it, nothing happened. When inspecting my code, I realised that I did not make a method for this process. To add this, I slightly changed my mousePressed method to allow the removal of notes. This involved checking if the user was holding the end of the note, and if they were, this meant that they wanted to resize the note. If they were not, this means that they were needing to remove the note, so the removeNote method is called. Here is the code for that section:

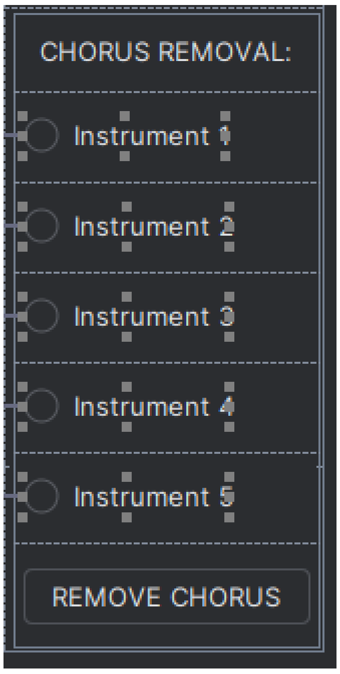


```
if(selectedNote != null) {
    if(col == selectedNote.getStart() + selectedNote.getLength() - 1) {
        resizing = true;
    }
    else{
        instrument.removeNote(selectedNote);
        selectedNote = null;
    }

}
```

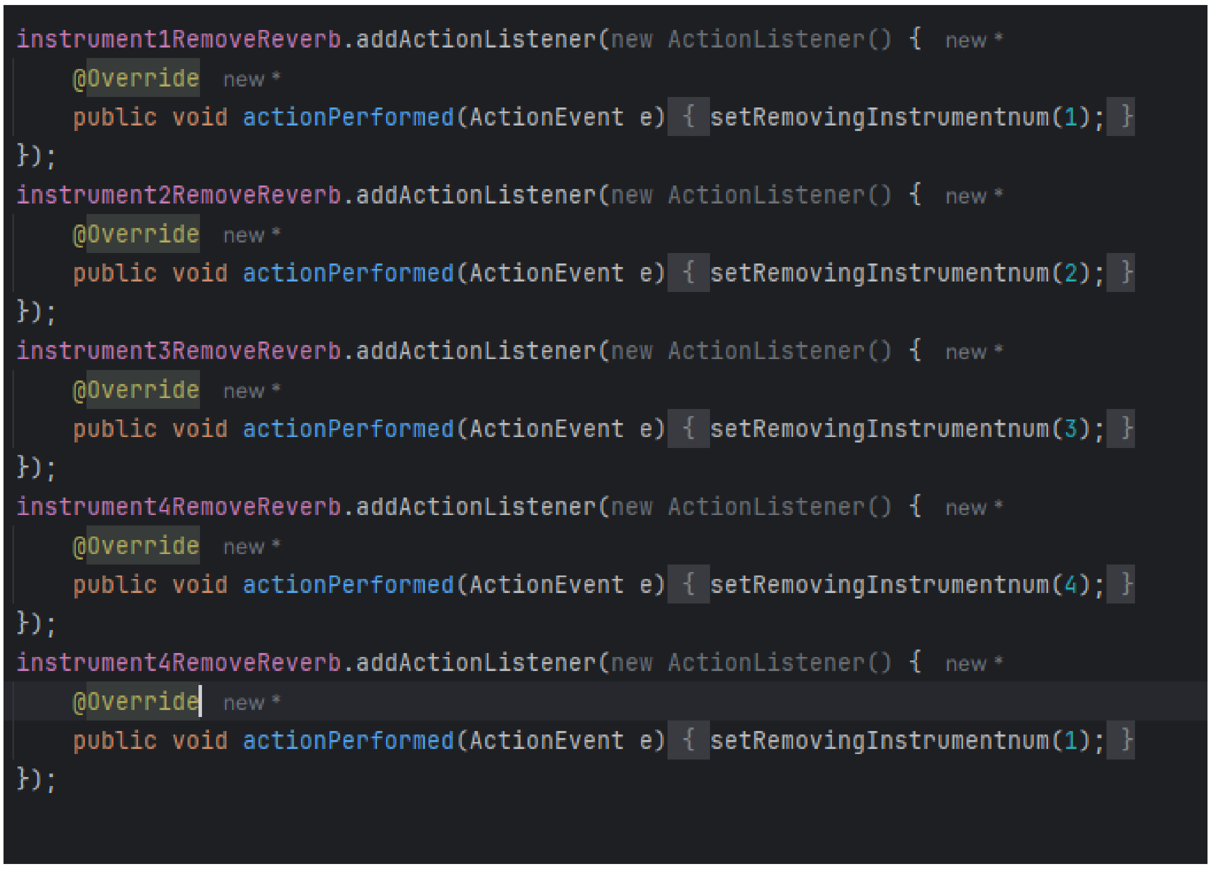
This code worked, so I was confident that I was complete with the resizing notes section. Since this section was complete, I moved onto creating a feature to remove effects from an instrument. This feature should be very simple and only involve creating a new section on the reverb and chorus form to decide what instrument they would like to remove the effect from, and then a button to remove that effect.

To start this, I created a new section of the reverb and chorus forms. This section involves an area to decide what instrument they would like to remove the effect from, and then a button to remove that effect. This means that the user will have a functional and simple interface for removing an effect if they wish to do so. Here is the layout for both the chorus and reverb effects removal section of their respective forms:

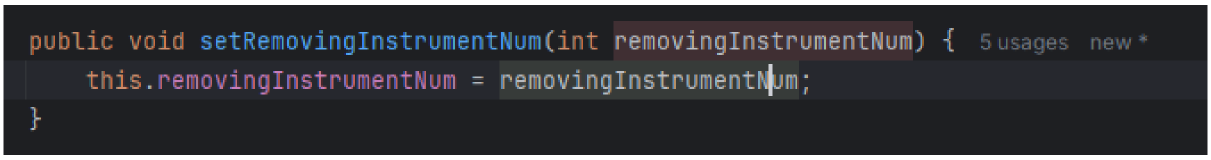




Now that the structure of these sections of the forms were created, I moved onto making the functionality behind selecting the instrument that the user wishes to remove the reverb effect from. This was simple as I had this logic in my pseudocode. This code will allow me to know what instrument the user wants to remove the reverb effect from, so that other code can run and effectively remove this effect when the reverbRemovalButton is pressed. Here is the code for that section:

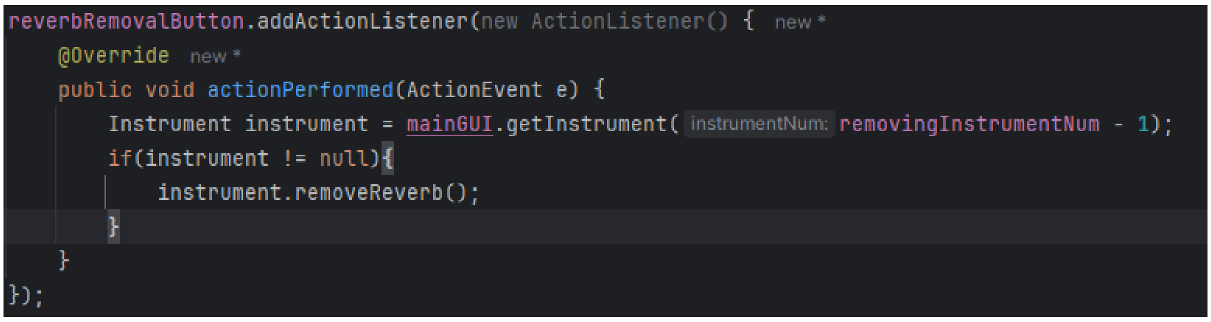


```
instrument1RemoveReverb.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentnum(1); }
});
instrument2RemoveReverb.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentnum(2); }
});
instrument3RemoveReverb.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentnum(3); }
});
instrument4RemoveReverb.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentnum(4); }
});
instrument4RemoveReverb.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentnum(1); }
});
```

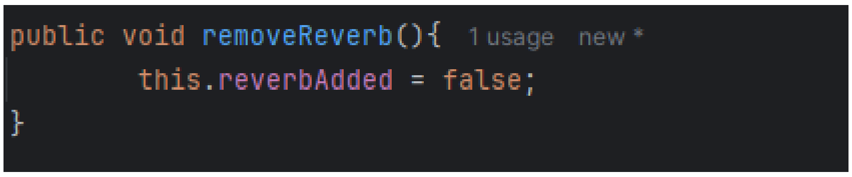


```
public void setRemovingInstrumentNum(int removingInstrumentNum) {  5 usages  new *
    this.removingInstrumentNum = removingInstrumentNum;
}
```

This code simply sets an attribute – removingInstrumentNum – to the number of the instrument that is selected in the radio buttons. This will allow me to create an action listener for the reverbRemovalButton which can remove the reverb effect from the correct instrument. This was what I completed next so that this feature had full functionality and the user could remove effects as they wish. This is the code for that section:

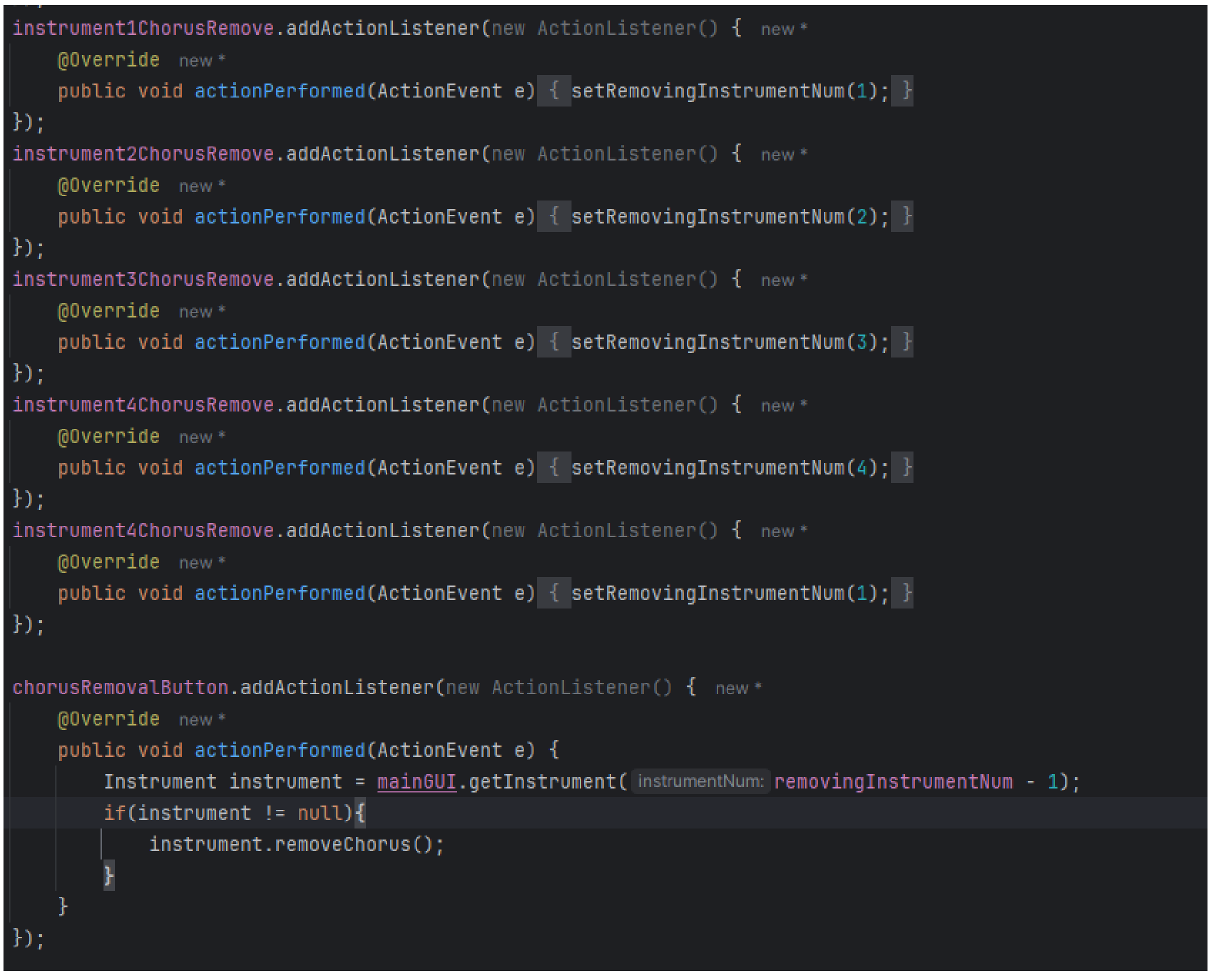


```
reverbRemovalButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument( instrumentNum: removingInstrumentNum - 1);
        if(instrument != null){
            instrument.removeReverb();
        }
    }
});
```

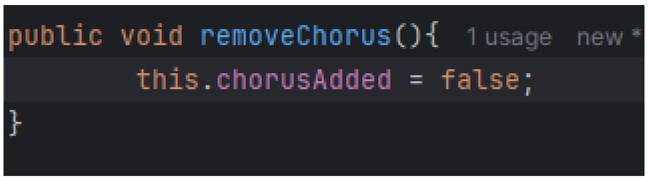


```
public void removeReverb(){  1 usage  new *
        this.reverbAdded = false;
}
```

The next stage was to do this for the chorus effect. This is a very simple process, as it has a similar logic to removing reverb. This means that I am able to copy the code I have already written for the removing reverb process and only have to change small bits. Since this was very simply, I will paste all code for this process here:



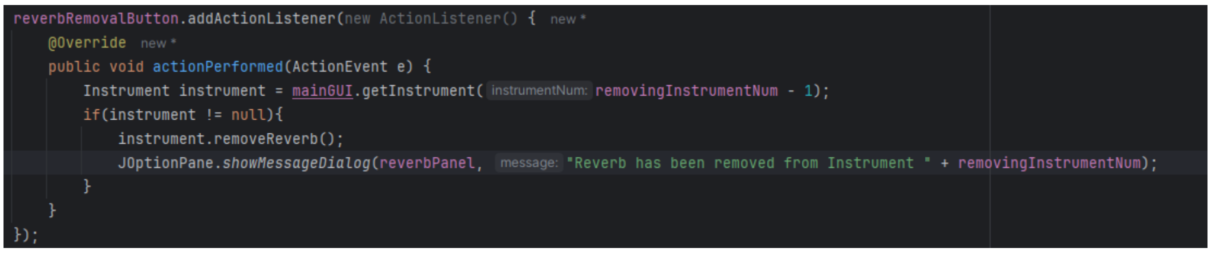
```
instrument1ChorusRemove.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentNum(1); }
});
instrument2ChorusRemove.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentNum(2); }
});
instrument3ChorusRemove.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentNum(3); }
});
instrument4ChorusRemove.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentNum(4); }
});
instrument4ChorusRemove.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) { setRemovingInstrumentNum(1); }
});

chorusRemovalButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument( instrumentNum: removingInstrumentNum - 1);
        if(instrument != null){
            instrument.removeChorus();
        }
    }
});
```



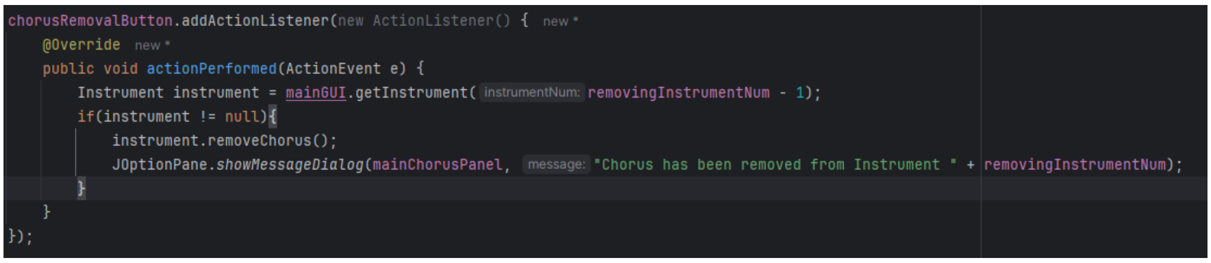
```
public void removeChorus(){  1 usage  new *
        this.chorusAdded = false;
}
```

I was confident that this feature was now fully functional so I did some tests on it, by adding an effect and then playing notes, before removing the effect and playing them again. In this process, I was observing if the desired effect had effectively been removed once I had asked it to do so.

From this test, I found that it did work, however it was not clear after pressing the "remove effect" that it had actually been removed without playing the sequence back. To fix this problem, I made code that displays a JOptionPane which clearly states that the effect has been removed. I could not remember how to use this feature, so I referred to geeksforgeeks (https://www.geeksforgeeks.org/java/java-joptionpane/) to remind me how to use JOptionPanes. This will make it clearer to the user that their desired action has actually taken place, without them having to play their sequence back. This therefore increases the usability of my program. Here is the code for that module:



```
reverbRemovalButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument( instrumentNum: removingInstrumentNum - 1);
        if(instrument != null){
            instrument.removeReverb();
            JOptionPane.showMessageDialog(reverbPanel,  message: "Reverb has been removed from Instrument " + removingInstrumentNum);
        }
    }
});
```



```
chorusRemovalButton.addActionListener(new ActionListener() {  new *
    @Override  new *
    public void actionPerformed(ActionEvent e) {
        Instrument instrument = mainGUI.getInstrument( instrumentNum: removingInstrumentNum - 1);
        if(instrument != null){
            instrument.removeChorus();
            JOptionPane.showMessageDialog(mainChorusPanel,  message: "Chorus has been removed from Instrument " + removingInstrumentNum);
        }
    }
});
```

I tested to see if this message did actually pop up when an effect had been removed, and it did. This means that the feature of removing an effect is fully functional, meaning that my program is more usable for people to change their minds over whether they would like an effect added to their sequence.

During research for learning how to zoom in and out of the sequencer, I realised that there is not a way to directly add this feature in through intelliJ idea. This means that this feature would be very complex to implement. Since this feature does not add to the functionality of my program I have deemed it as unnecessary and will not be adding it during this stage. This does impact the accessibility of my program, however it is not necessary for most users, so it will not be implemented.

<u>Testing and Analysis:</u>

Test numbe r	Description of test	Test data	Expected outcome	Achieved?
1	Test that when the note is resized, this is visualised on the sequencer	Add a note, then resize it multiple times for different lengths	The note should resize when it had been dragged	**YES**
2	Test that a resized note has a different playtime length	Add a note, play it, change its length, play again, repeat for different lengths	The note should have a different duration based on how long it is shown to be	**YES**
3	Test effects working with different note durations	Add a note, play it, add reverb or chorus effect, play again, change duration, repeat	The effect should work properly (i.e. play the effect after the start of the note) with different durations	**YES**
4	Test the removal of effects is functional	Add notes, add effects, play the sequence, remove the effects, play the sequence again	The effects should be added and removed after they have been asked to do so	**YES**

5	Test the display of an effects removal method	Add and remove an effect	A JOptionPane should display that the effect has been removed	**YES**

Analysis:

I am disappointed that I am not able to create a way of zooming into the sequence. At first, I thought that this feature would be very simple to implement, however during development I realised that it would be a very time consuming and complex process. This means that I will not implement this feature so that I can meet my deadline for this project. This feature is not necessary for the functionality of my program so is not a large problem, however would have been useful to increase the accessibility to it. This does however mean that I am able to ignore this feature to make the development process achievable in the given time frame.

I am very happy with how the note duration feature works, since this is vital for the usability of my program. This feature is useful for the user to be able to add more depth and complexity into their sequences. It works perfectly as expected too, so I have no errors with this feature that I will have to fix. This means that the user is able to use whatever note durations they would like in theit sequence, therefore providing more functionality to them.

Additionally, the effects removal system works as expected too. This means that the effect is effectively removed and a message displays that the effect has been removed. This means that the user is able to clearly see that the effect has been removed, when it functionally has. This means that the user can make better decisions about how they would like their sequence to sound and therefore makes my program more user-friendly.

Stage 4 – Export System:

In this stage, the only feautre that I plan to implement in this stage in the ability to export your final piece as an mp3 file. This feature is quite essential so that the user is able to make sure that they can have a file to play their music in an interface outside of the sequencer. The reason that I am completing this stage now is because this feature relies on the sequencer having full functionality, so that a full piece can be exported.

I am not yet able to code these features in with the knowledge that I currently have. This means that I will have to research how to export a file using Intellij in java, and learn how to write certain noises into this file. I will cite any of the websites and resources that I use to do this process.

At the end of this stage, the user should be able to sequence a full piece of music and then export it as an mp3 file. This mp3 file should include everything that the user has included in their piece, which may include: all notes at the correct duration, all effects included with the correct attributes, and all instruments.

During research for this stage an completiong, I realised that converting the final sequence into an mp3 file would take a lot of complex and unnatural coding. I found that converting the final sequence into a WAV file would be a lot easier. Another benefit of this change is that when doing stakeholder requirement research, one stakeholder stated that WAV files are a lot more usable for music producers. This means that not only will the coding be less complex and time consuming, but also more usable and functional for my users. This research was done across these websites:

- https://forum.processing.org/two/discussion/4339/how-to-save-a-wav-file-using-audiosystem-and-audioinputstream-of-javasound.html
- http://docs.oracle.com/javase/8/docs/technotes/guides/sound/programmer_guide/chapter7.html
- https://thiscouldbebetter.wordpress.com/2011/08/14/reading-and-writing-a-wav-file-in-java/
- http://srcrr.com/java/oracle/openjdk/6/reference/com/sun/media/sound/SoftSynthesizer.html

Design:

Algorithm design:

- Below is the pseudocode for rendering the final sequence of the notes array into a midi sequence. This code will be added into my instrument class. In this method, the note is inspected to see its pitch, where is starts, where is ends and what sound to use, before using this information to build a MIDI file in memory. This is very simplified pseudocode and will be more complex during development, however the main idea of the method is there:

```
Public sequence buildMIDI(){
    Sequence sequence = new sequence();
    Track track = sequence.createTrack();

    FOR Note currentNote in notes {
        Int pitch = currentNote.getPitch();
        Int start = currentNote.getStart();
        Int end = currentNote.getStart() + currentNote.getLength();

        Track.add(new MidiEvent(on, start);
        Track.add(new MidiEvent(off, end);
    }
    Return sequence;
}
```

- The next pseudocode that I will create is the code for finding the java's build in MIDI synthesizer that can play MIDI notes straight into a WAV file. This is done by checking all MIDI devices on your memory until it finds one that supports the rendering of sound directly into memory. This was a particularly complex idea before I did my research, however JavaTips (https://www.javatips.net/api/com.sun.media.sound.audiosynthesizer) helped me to grasp this, and better understand what I should do for this. The reason for this method is so that I have a suitable method for creating a WAV file rather than simply playing the sequence. Here is the simplified pseudocode:

```
Private audioSynthesizer getAudioSynthesizer(){
        midiDeviceInformations[] = getMidiDeviceInfo();
        FOR midiDeviceInformations{
            device = getMidiDevice(midiDeviceInformation)
            IF device.isRenderer() THEN
```

```
                Return device;
            ENDIF
        }
        Return null;
    }
```

- The next pseudocode that I will paste is the pseudocode for actually exporting the WAV file. In this pseudocode, the sequence will be built as a midi version using the first pseudocode method that I created. It will then play the sequence while rendering it rather than into through speakers. This silently recorded audio is then written into memory as a WAV file. This is a very complex method, and will be added to during development, so the upcoming pseudocode is a very simplified version of what will actually be produced. I used JavaTips (https://www.javatips.net/api/com.sun.media.sound.audiosynthesizer) again to help me with the logic for this method:

```
Public void exportToWav(){
    Sequence sequence = buildMidiSequence();
    Synth = findAudioSynthesizer();
    AudioInputStream stream = synth.openStream(format, info);

    Soundbank soundbank = getDefaultSoundbank();

    Sequencer.open();
    Sequencer.start();

    WHILE sequencer.isRunning()
        Sleep(20)
    ENDLOOP
```

```
        Sequencer.stop();
        Sequencer.close();


        AudioSystem.write(stream, WAVE, output);


        Synth.close();
}
```

- That is all of the pseudocode for this section, other than the code for a Jbutton to run these exporting methods. I will not create pseudocode for this, since it is a relatively simple method and can be done very easily without necessary pseudocode. This means that I am finished with pseudocode and can move onto creating the data dictionary, which for this section is very small, since only very small yet complex methods are added during this section.

Data:

- Below I will create a data dictionary to hold all the attributes and methods that are necessary for development in this stage. This will make it easier during development, since it will allow me to refer to it to better understand what my attributes and methods purposes are and the type that they return

Attributes:

- There are no attributes that are being introduced in this stage, so I have no necessary attribute data dictionary for this stage. This means that I have no requirement for this smaller section. Despite making me have more time for development, this stage requires a lot of research and finding new ways of coding that I have not previously been doing. This means that this extra programming. time will be necessary.

Methods:

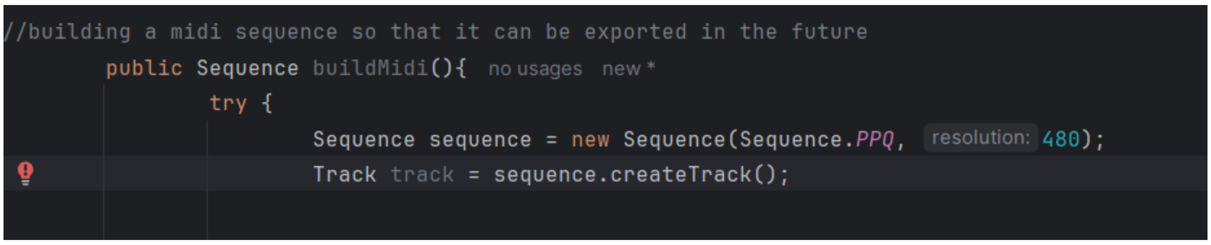
Name	**Return type**	**Description**	**Held in class:**
buildMIDI	Sequence	Will build a midi sequence so that	Instrument

		it can be played into memory	
getAudioSynthesizer	audioSynthesizer	Getter for a audiosynthesizer in memory which can render audio silently rather than playing is through speakers	Instrument
exportToWav	Void	uses the sequence built in buildMIDI in an audioSynthesizer to render the audio into a file, before exporting this file	instrument

Development:

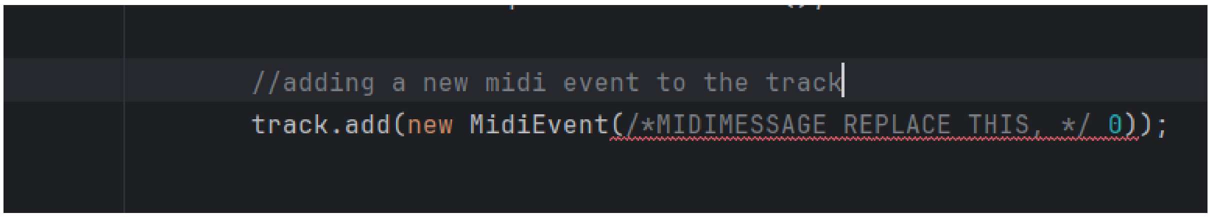
- Before I started to implement the first method – buildMidi – I looked for a youtube tutorial so that I could better understand the processes that I will have to complete in this stage. I found this video - https://www.youtube.com/watch?v=CaqpL4r2lZE – which clearly explained how to create a MIDI sequence in java. Alongside the other research that I previously mentioned, this will help me to build a midi sequence in order to export it later on.
- To start with this method, I first declared the method, the sequence, and the track that is created through this sequence. This means that I will be able to add the notes into this track. This will mean that I am able to build a midi track which can later be played silently into a sound rendering device, so that it can later be created as a WAV file format. This means that this process is essential for the rest of the exporting process to happen. During the creation of this code, I had to find out what I must pass into the constructor for creating a sequence. To help me with this I looked at Oracle documentation - https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/sound/

midi/Sequence.html#:~:text=numTracks)%20throws%20InvalidMidiDataException-,Constructs%20a%20new%20MIDI%20sequence%20with%20the%20specified%20timing%20division,specified%20in%20ticks%20per%20beat. Here is the code for the beginning of that method within the instrument class:



```
//building a midi sequence so that it can be exported in the future
        public Sequence buildMidi(){  no usages  new *
                try {
                        Sequence sequence = new Sequence(Sequence.PPQ,  resolution: 480);
                        Track track = sequence.createTrack();
```

- The next step for this method was to add a midi event to the track. To do this, I again referred to Oracle documentation - Track (Java SE 11 & JDK 11) and https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/sound/midi/MidiEvent.html . This helped me to know what to pass into relevent methods that I will use. I struggled ot understand what I should pass in when creating a new MidiEvent. This took me a lot of time, so for now I will leave this out and come back to it. I will do this so that I have more time to think and research what variables I should pass in. This will hopefully give my program more infrastructure. Here is the broken up code:



```
            //adding a new midi event to the track
            track.add(new MidiEvent(/*MIDIMESSAGE REPLACE THIS, */ 0));
```

- After that, I had to create a for loop which cycles through all of the notes in a sequence, before adding them to the track. This is a useful for loop, since if it were not here, the track would be empty, and the exported WAV file will contain no sounds. Before the note is added to the track, I have to use getters within the note class to gather the pitch, start and end of each note, so that it can be added into the correct place within the track. Without this method, all of the notes within the track would be muddled in terms of pitch and timing. This means that this process is vital for the exported file to play properly. Here is the code for that:

- As you can see, there is no code underneath this explaination of what I should do. This is because I struggled with the addition of the correct notes into the track. I realised at this point that this feature is very complicated to implement, and far above my skill level of coding. Even with hours of research I do not think I will be able to implement this whole feature correctly within my given timespan.

- Additionally, I thought further about the necessity of an exporting feature and do not think that it is as necessary as I first thought. The only reason that my user would need to export their final piece, is if it were to be published, and since my program is aimed to more beginner musicians in secondary school and college, I do not think that it is a necessary feature for them. If they really desire to use a sequenceing program for the purpose of publishing music, there are more proffesional and advanced sequencers for this purpose.

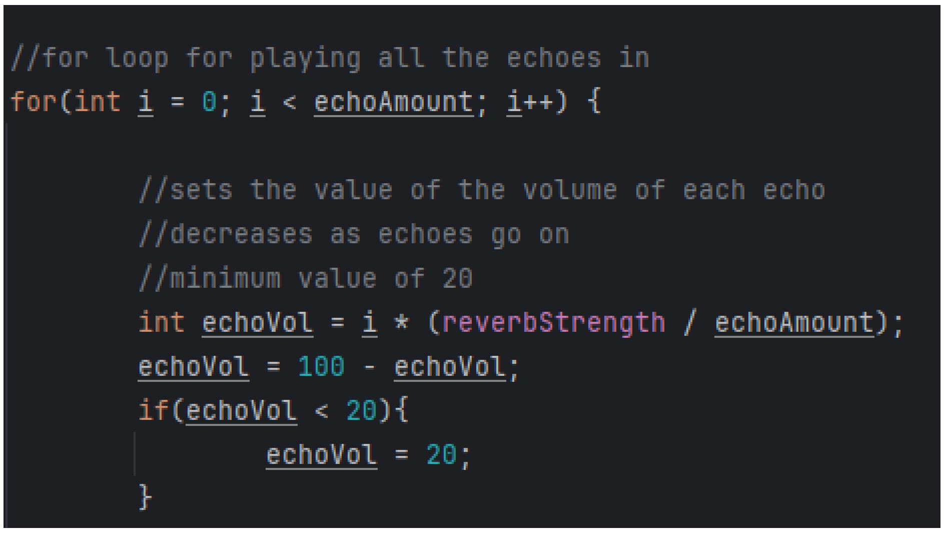
- For both of the reasons above, I have deemed this feature as discardable, meaning that I will not be adding this feature into my final project. I feel like this extra time is necessary for testing and cleaning up my code so that it is as functional and useable as possible.

- Before I can move onto cleaning up and commenting on my code, I first have to remove the code that I added during the small amount of development time in this stage. This involves removing the code that I added, and removing other parts of the GUI (such as the item in the toolsComboBox) that are linked to the addition of a exporting feature. I will not paste proof of this, since it is just empty space.

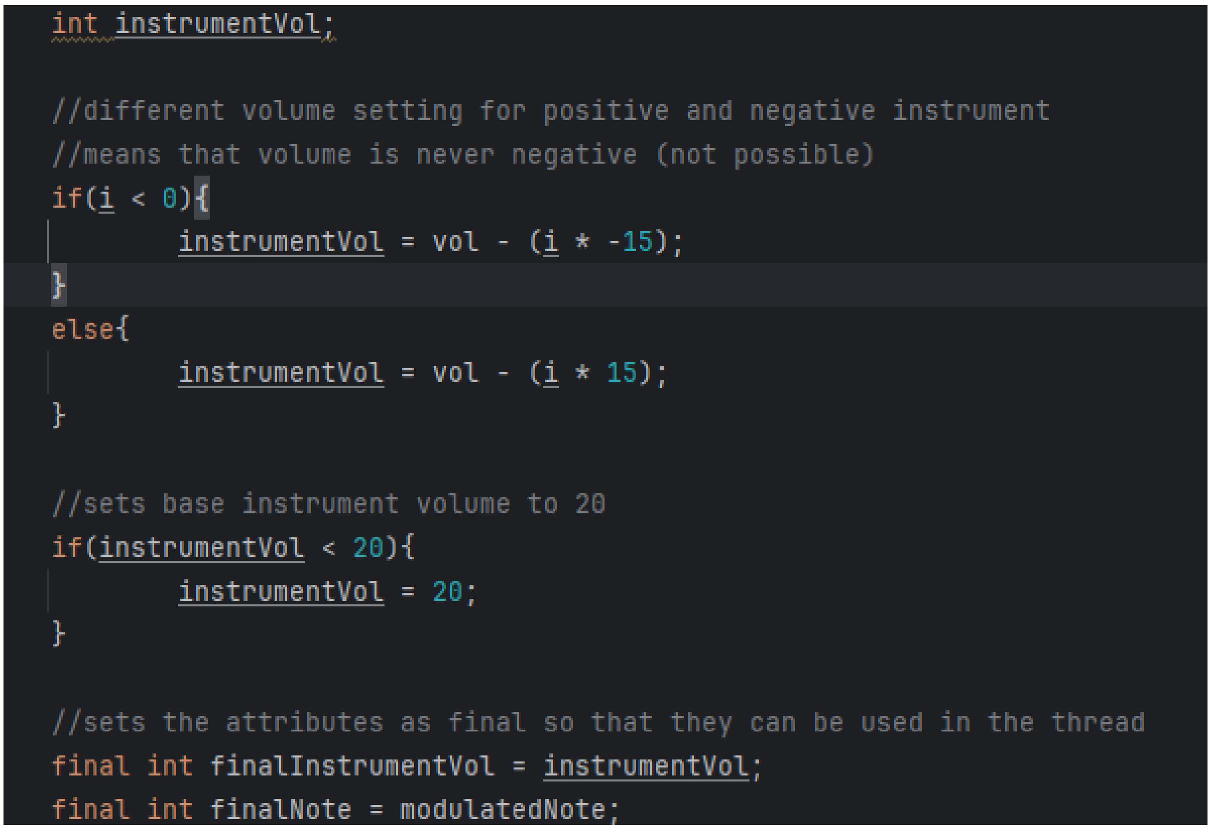
Evaluation:

- This stage did not go to plan at all. I have not added any feature to export your final piece in any format. I have explained my reasoning for this above in the deveopment section – I simply do not have enough time for such a complex and somewhat unnecessary stage. Therefore there is not much to evaluate in this stage, since I added nothing.

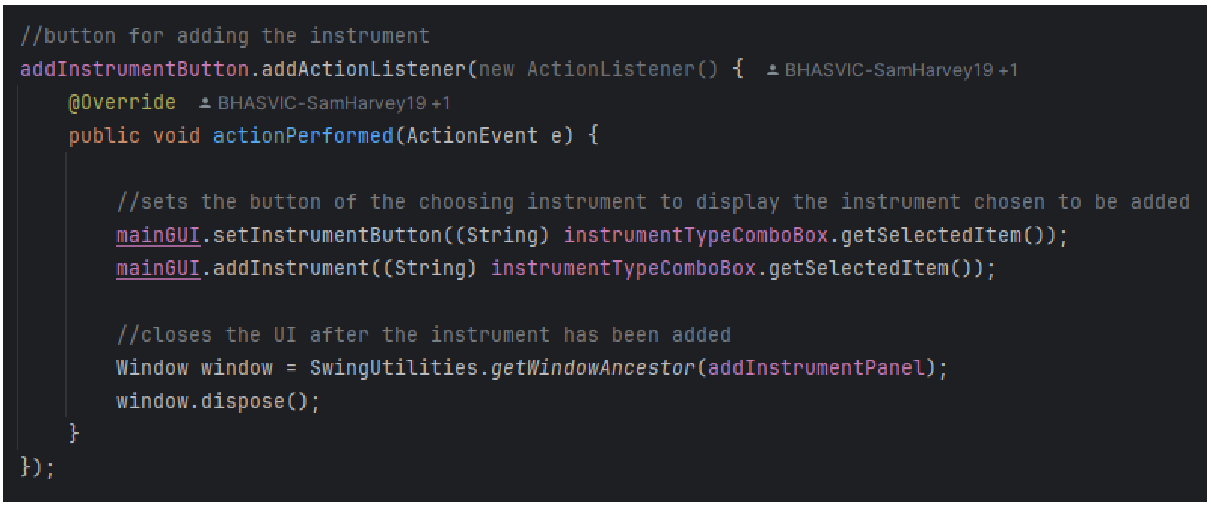
- Before I deemed my development as finished, I had to go through my code and add comments to display what certain areas of my code are used for. This will help me during testing so that I can easier see what part of my program is causing errors if any occur. I will not paste all of the comments that I add, since there will be so many, however I will show a small amount of these comments as proof that they have been added. Here are some comments that I added into my code:



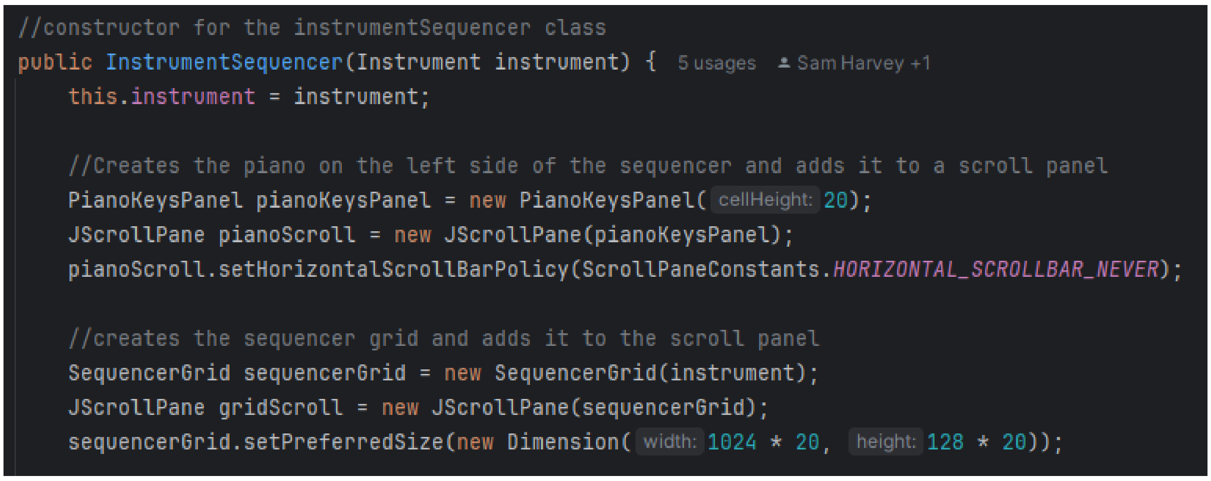
```
//for loop for playing all the echoes in
for(int i = 0; i < echoAmount; i++) {

        //sets the value of the volume of each echo
        //decreases as echoes go on
        //minimum value of 20
        int echoVol = i * (reverbStrength / echoAmount);
        echoVol = 100 - echoVol;
        if(echoVol < 20){
                echoVol = 20;
        }
```



```
   int instrumentVol;

   //different volume setting for positive and negative instrument
   //means that volume is never negative (not possible)
   if(i < 0){
           instrumentVol = vol - (i * -15);
   }
   else{
           instrumentVol = vol - (i * 15);
   }

   //sets base instrument volume to 20
   if(instrumentVol < 20){
           instrumentVol = 20;
   }

   //sets the attributes as final so that they can be used in the thread
   final int finalInstrumentVol = instrumentVol;
   final int finalNote = modulatedNote;
```



```
//button for adding the instrument
addInstrumentButton.addActionListener(new ActionListener() {  BHASVIC-SamHarvey19 +1
    @Override  BHASVIC-SamHarvey19 +1
    public void actionPerformed(ActionEvent e) {

        //sets the button of the choosing instrument to display the instrument chosen to be added
        mainGUI.setInstrumentButton((String) instrumentTypeComboBox.getSelectedItem());
        mainGUI.addInstrument((String) instrumentTypeComboBox.getSelectedItem());

        //closes the UI after the instrument has been added
        Window window = SwingUtilities.getWindowAncestor(addInstrumentPanel);
        window.dispose();
    }
});
```



```
//constructor for the instrumentSequencer class
public InstrumentSequencer(Instrument instrument) {  5 usages  Sam Harvey +1
    this.instrument = instrument;

    //Creates the piano on the left side of the sequencer and adds it to a scroll panel
    PianoKeysPanel pianoKeysPanel = new PianoKeysPanel( cellHeight: 20);
    JScrollPane pianoScroll = new JScrollPane(pianoKeysPanel);
    pianoScroll.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

    //creates the sequencer grid and adds it to the scroll panel
    SequencerGrid sequencerGrid = new SequencerGrid(instrument);
    JScrollPane gridScroll = new JScrollPane(sequencerGrid);
    sequencerGrid.setPreferredSize(new Dimension( width: 1024 * 20,  height: 128 * 20));
```