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];
}
```

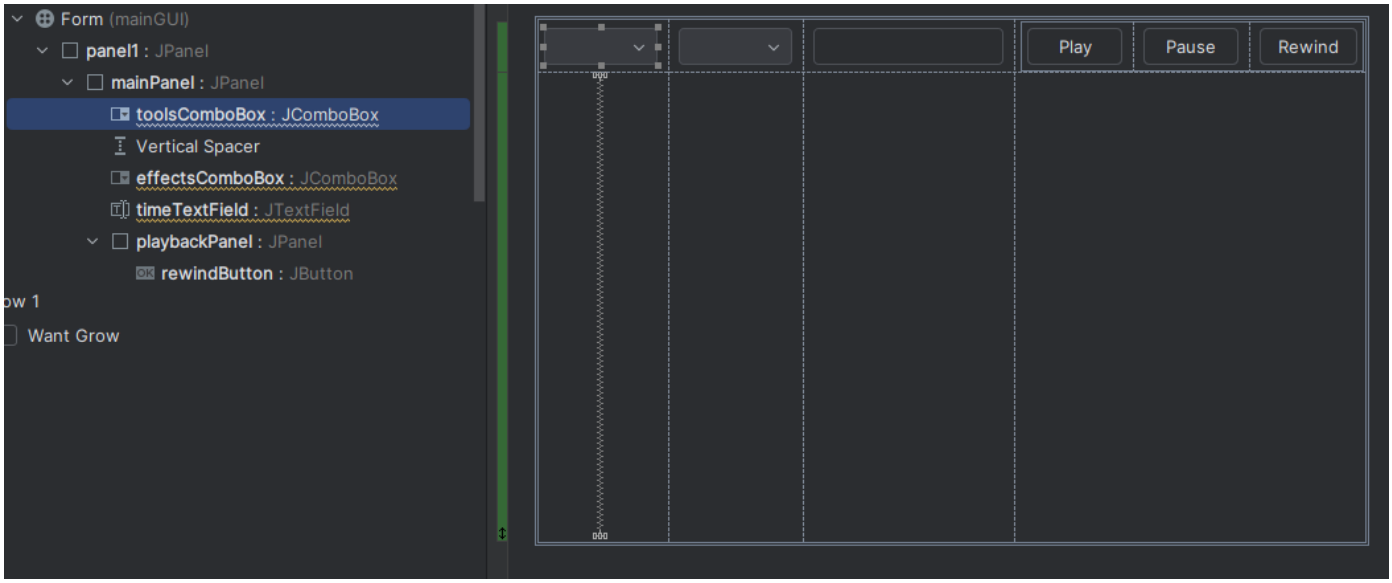
After that, I started creating the play method. For this I had to learn how to create and manipulate sound files, so I looked at REINTECH (https://reintech.io/blog/java-midi-programming-creating-manipulating-midi-data), and rememberjava (https://rememberjava.com/midi/2017/01/13/midi_basics.html) which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

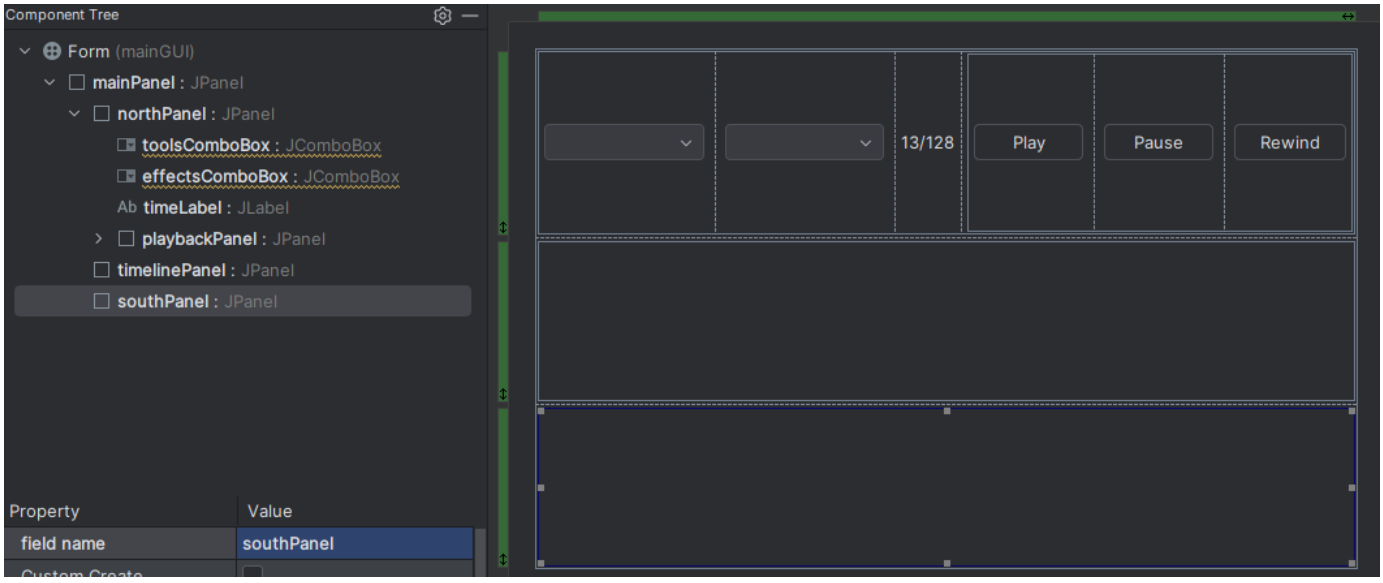
This is only a draft of the final code and may change based on other parts that I code (E.g. changing tempo or multi-sequencing). This method is wrapped in a try block so any errors can be caught. The first thing the play method does is creating a MIDI

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.

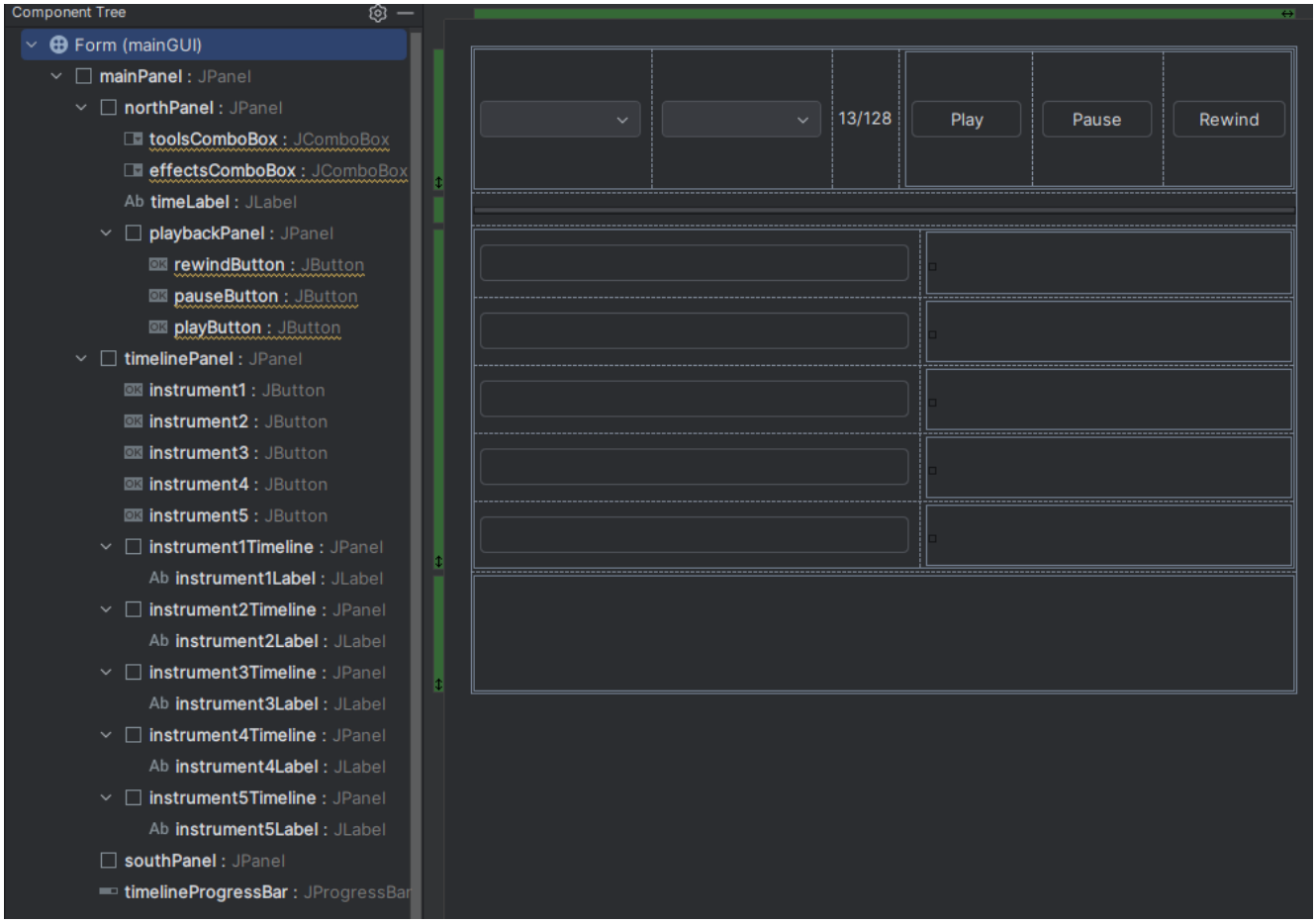
The next thing that I constructed was the main sequencer GUI. I did this my adding several JButtons, JComboBox's, and JTextField's onto a JPanel on my mainGUI. I gave these relevant names and places on the GUI with respect to my initial GUI designs. I did this so that there is a main navigation GUI for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



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



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



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

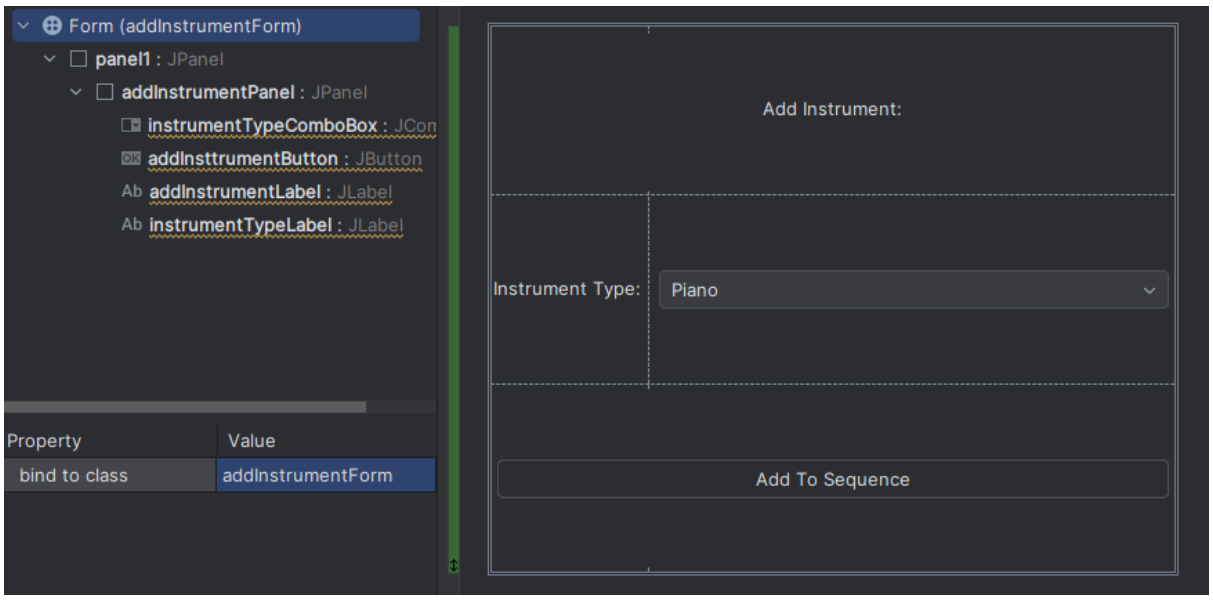
```
Instrument instrument = new Instrument( type: 3);
instrument.addNote( note: 64,  time: 1);
instrument.addNote( note: 64,  time: 2);
instrument.addNote( note: 64,  time: 3);
instrument.addNote( note: 64,  time: 4);
instrument.addNote( note: 64,  time: 5);
instrument.addNote( note: 64,  time: 6);

instrument.play();
```

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

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

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

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

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

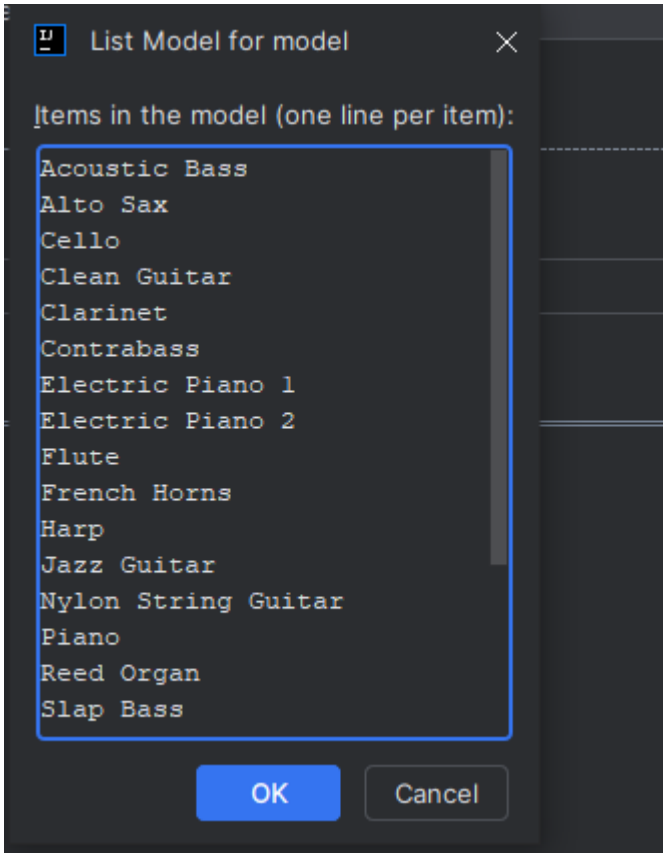


```
import javax.swing.*;

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:



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:

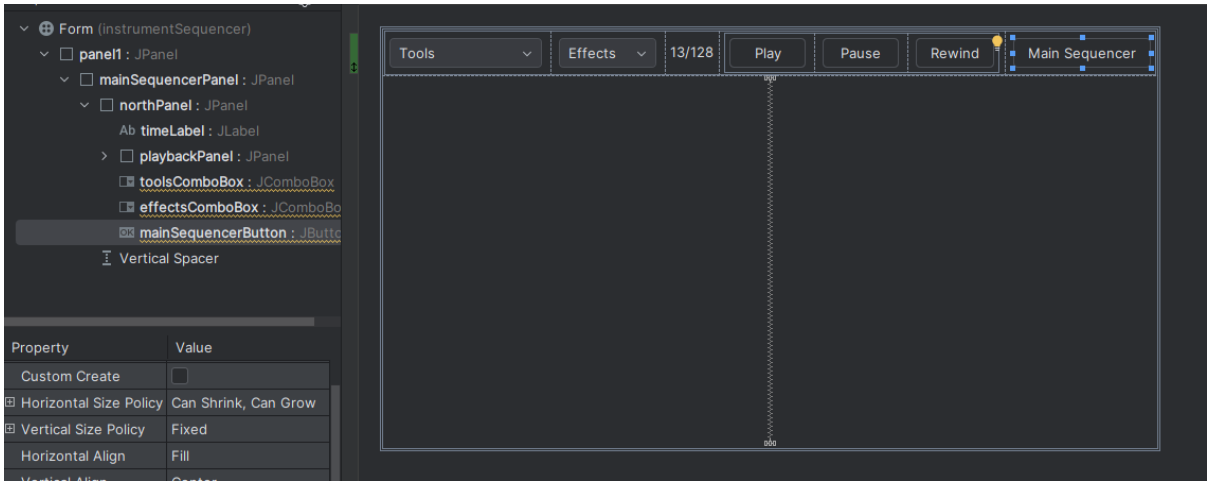
```
public addInstrumentForm() {  new *
    instrumentTypeComboBox.addActionListener( ActionEvent e -> {
        }else if("Nylon String Guitar".equals(instrumentTypeComboBox.getSelectedItem())) {
            chosenInstrument = 24;
        }else if("Steel String Guitar".equals(instrumentTypeComboBox.getSelectedItem()))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



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

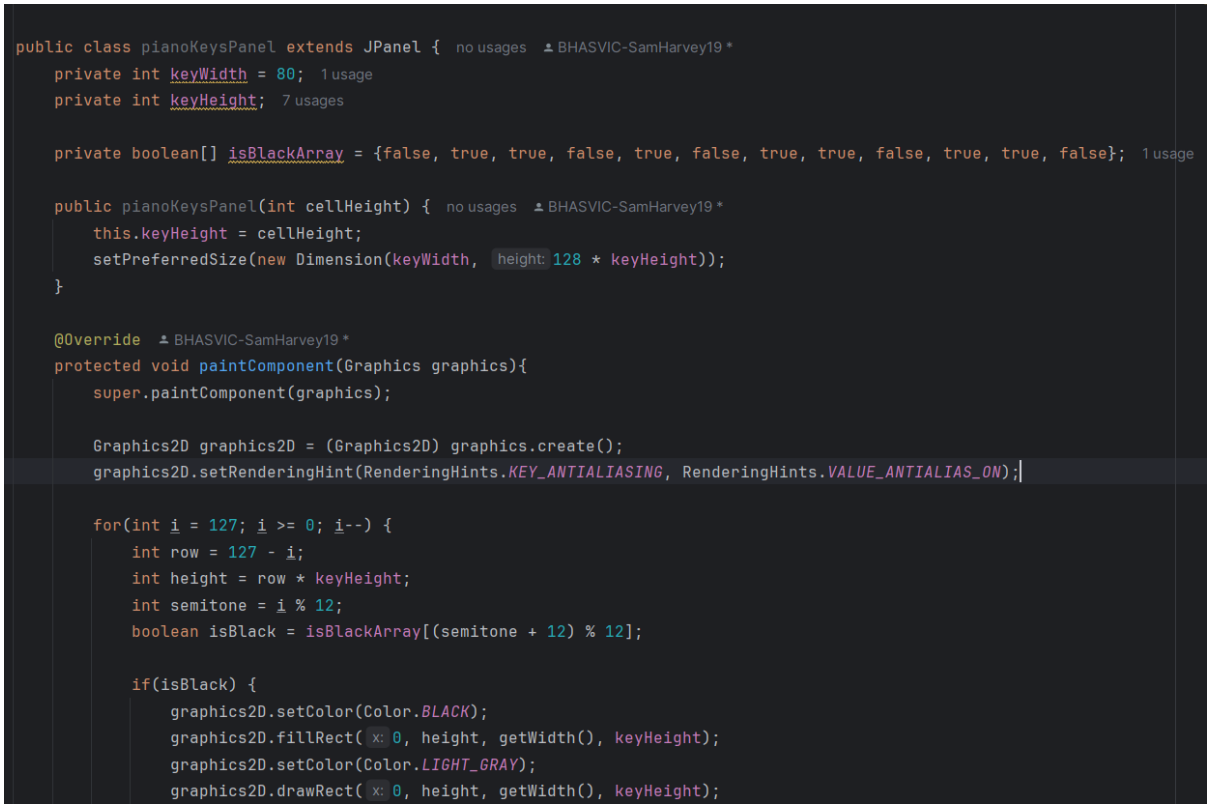
instruments in an array so that they can all be accessed through one array. Here is all the code that I changed/added:

```
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:



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

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

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

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

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

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

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

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

Next, I had to make this component visible on the instrumentSequencer, so that the piano could be seen and utilised by the user. I also needed to make the sequencer able

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

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

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

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

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

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

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

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

specific instrument when a cell is clicked. Here is the code:

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


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

    private Instrument instrument;  no usages
}
```

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

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


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

    private Instrument instrument;  1 usage

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



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

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

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

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

                }
            }
        }
    }
}
```

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

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

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

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

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

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

```
this.instrument = instrument;

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

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

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

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

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

```
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);
```

```
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() {
    @Override
    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() {
    @Override
    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				

2				
3				
4				
5				
6				