Tampere University of Applied Sciences

5G00DM69-3004 Graphical User Interfaces

Part 2: JavaFX, Weekly exercises

Jingjing Yang

March-April 2024

Bachelor's Degree Programme in Software Engineering

CONTENTS

1 Exercise 1

1.1 Functionalities

- The application includes a text field for editing the job name with an accompanying label.
- There's a button that updates the job with the new name entered in the text field.
- The application updates the job instance in two scenarios:
  - When the user presses Enter while the keyboard focus is in the text field.
  - When the user clicks the Update button.
- The application validates the job name entered by the user to ensure it's not empty, has no leading or trailing spaces, and is not longer than 20 characters.
- The application prints either a text representation of the updated job or an error to the standard output after each update. The text representation of the updated job includes the job's name, starting note, ending note, and interval value.

1.2 Solution

- The application uses JavaFX for building the interface.
- The text field for editing the job name is created using the TextField class in JavaFX.
- The accompanying label for the text field is created using the Label class in JavaFX.
- The HBox class is used to group the label and the text field horizontally in the same box.
- The VBox class is used to arrange the HBox and the update button vertically in the same box.
- The setOnAction method is used with a lambda expression to handle the action events of pressing Enter while the text field has keyboard focus and clicking the Update button. This method is used to update the job instance.
- The entered job name is validated by checking its length and whether it's empty or not. This is achieved using conditional statements in the updateJobName method.
- The application prints to the standard output using System.out.println. It either prints a text representation of the updated job or an error message based on the validation of the job name.

I add a setName method to the Job class (found on Moodle) to allow for the job name's modification.

1.3 Job.java v1

```
import java.util.UUID;

import java.util.List;

import java.util.ArrayList;
```

```
/**
 * Describes a sampling job with successive notes.
 */
public class Job {
    /**
     * Interval between notes in semitones.
     */
    public enum Interval {
        ONE(1),
        THREE(3),
        SIX(6),
        TWELVE(12);

        private final int value;

        private Interval(int value) {
            this.value = value;
        }

        /**
         * Gets the ordinal value of this enum instance.
         */
        public int getValue() {
            return value;
        }
    }

    /**
     * Constructs a job with default values.
     */
    public Job(String name) {
        this.name = name;
        this.id = UUID.randomUUID();
        this.fromNote = 40;
```

```
    this.toNote = 120;
    this.interval = Interval.SIX;
}

/**
 * Gets the name of the job.
 */
public String getName() {
    return this.name;
}

/**
 * Gets the unique identifier of the job.
 */
public UUID getId() {
    return this.id;
}

/**
 * Sets the name of the job.
 */
public void setName(String name) {
    this.name = name;
}

/**
 * Gets the first note of the range.
 */
public int getFromNote() {
    return this.fromNote;
}

/**
 * Sets the first note of the range.
 */
```

```
public void setFromNote(int note) {
    if (note < 0 || note > 127) {
        throw new IllegalArgumentException("Note must be 0...127");
    }
    this.fromNote = note;
}

/**
 * Gets the last note of the range.
 */
public int getToNote() {
    return this.toNote;
}

/**
 * Sets the last note of the range.
 */
public void setToNote(int note) {
    if (note < 0 || note > 127) {
        throw new IllegalArgumentException("Note must be 0...127");
    }
    this.toNote = note;
}

/**
 * Gets the interval between the notes.
 */
public Interval getInterval() {
    return this.interval;
}

/**
 * Sets the interval between the notes.
 */
public void setInterval(Interval i) {
```

```
    this.interval = i;
}

public List<Integer> getNotes() {
    List<Integer> notes = new ArrayList<>();

    int note = this.fromNote;
    while (note <= this.toNote) {
        notes.add(note);
        note += this.interval.getValue();
    }

    return notes;
}

/**
 * Gets a string representation of the job.
 */
@Override
public String toString() {
    return String.format(
        "%s: from %d to %d by %d semitones",
        this.name,
        this.fromNote,
        this.toNote,
        this.interval.getValue()
    );
}

//
// Private fields
//

private String name;
private UUID id;
```

```
    private int fromNote;
    private int toNote;
    private Interval interval;
}
```

1.4 TestJob.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TestJob extends Application {
    private Job job;  // Job instance

    @Override
    public void start(Stage stage) {
        job = new Job("Test");  // Initialize a new Job
        job.setFromNote(60);
        job.setToNote(92);

        // Create HBox for job name input
        HBox jobNameInput = new HBox();

        // Create TextField for job name
        TextField jobNameField = new TextField(job.getName());
        jobNameField.setPromptText("Enter Job Name");  // Text field to enter job name
        jobNameField.setPrefWidth(300); // Set the preferred width to 300 pixels

        // Set text field to trigger job name update when Enter key is pressed
```

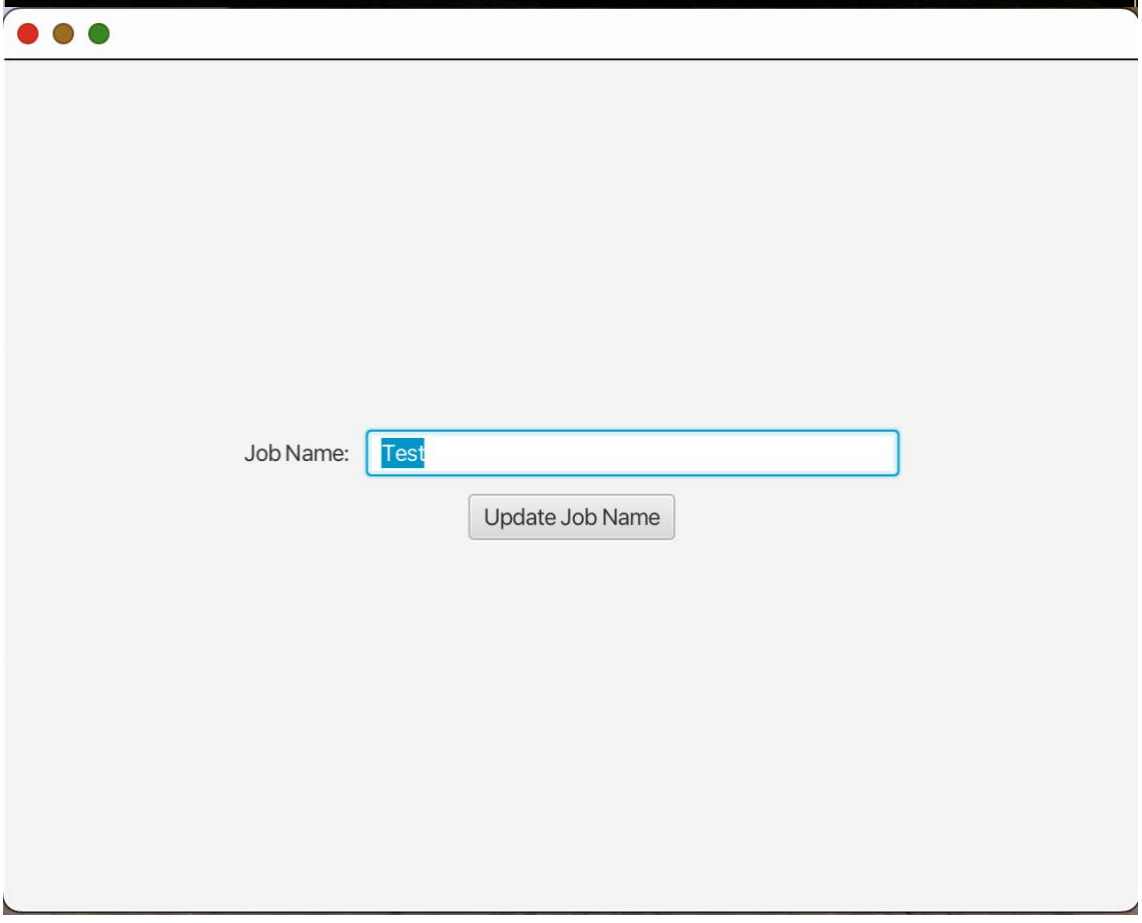
```
        jobNameField.setOnAction(e -> updateJobName(jobNameField));

        // Create Label for job name
        Label jobNameLabel = new Label("Job Name:");  // Label for the text field

        // Add label and text field to the HBox
        jobNameInput.getChildren().addAll(jobNameLabel, jobNameField);  // Add both to HBox
        jobNameInput.setSpacing(10);  // space between nodes
        jobNameInput.setAlignment(Pos.CENTER);  // Center the input


        // Create new VBox layout
        VBox layout = new VBox();

        // Create Button for update
        Button updateButton = new Button("Update Job Name");  // Update button
        // Set button to trigger job name update when clicked
        updateButton.setOnAction(e -> updateJobName(jobNameField));

        // Add components to layout
        layout.getChildren().addAll(jobNameInput, updateButton);
        layout.setSpacing(10); // space between jobNameInput and updateButton
        layout.setPadding(new Insets(10));
        layout.setAlignment(Pos.CENTER);  // Center the layout

        // Create Scene
        Scene scene = new Scene(layout, 640, 480);
        stage.setScene(scene);
        stage.show();
    }

    private void updateJobName(TextField jobNameField) {
        String newName = jobNameField.getText().trim();  // Trim leading and trailing whitespaces

        // Check if job name is not empty and not exceeding 20 characters
        if (newName.length() > 0 && newName.length() <= 20) {
```

```
            job.setName(newName);  // Set the new job name
            System.out.println("Updated Job: \n\t" + job);  // Print the updated job
        } else {
            System.out.println("Invalid Job Name. It should be 1-20 characters long.");  // Print error
if invalid
        }
    }

    public static void main(String[] args) {
        launch();  // Launch the application
    }
}
```

1.5 Testing

1. Check the initial display of the application.

```
jingjingyang@mac ex1 % ls
Job.java        TestJob.java
jingjingyang@mac ex1 % javac --module-path $PATH_TO_FX
--add-modules javafx.controls TestJob.java
jingjingyang@mac ex1 % ls
Job$Interval.class      Job.java                TestJob
.java
Job.class               TestJob.class
jingjingyang@mac ex1 % java --module-path $PATH_TO_FX -
-add-modules javafx.controls TestJob.java
```

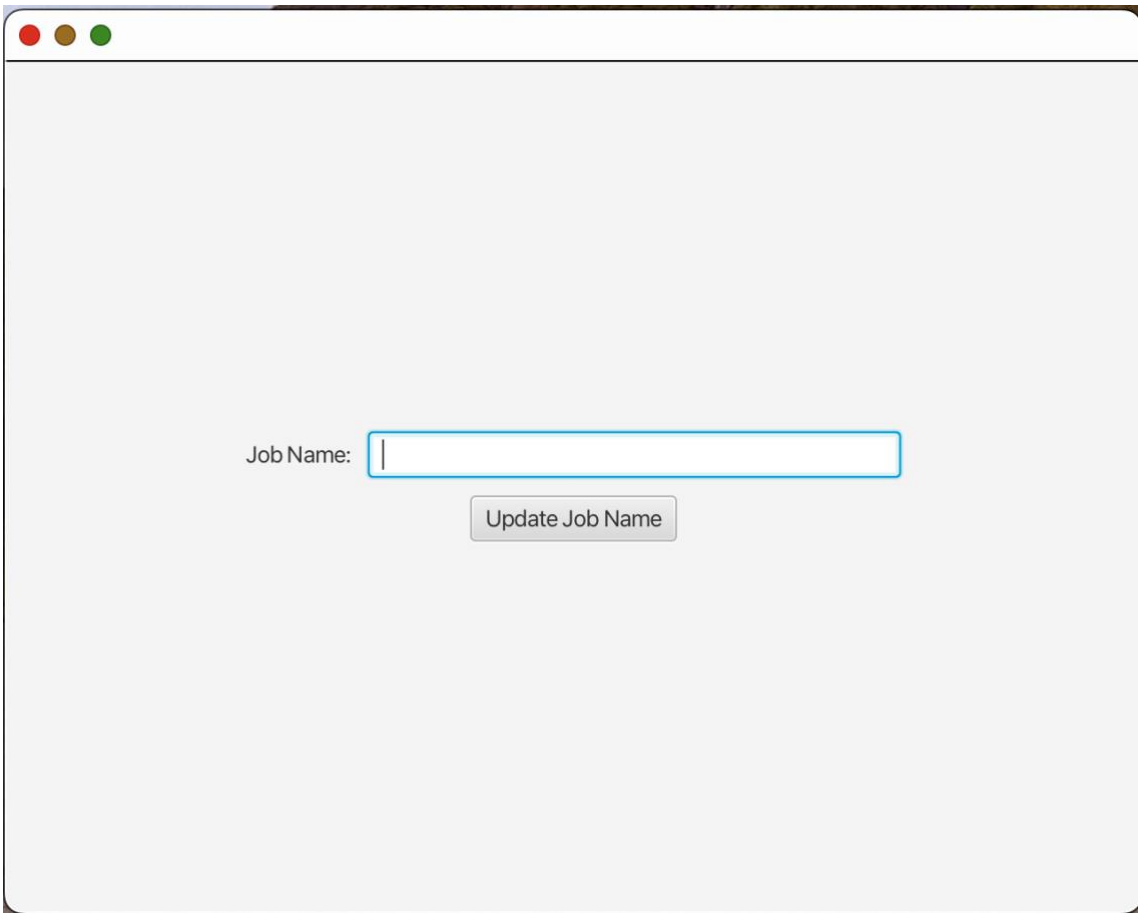


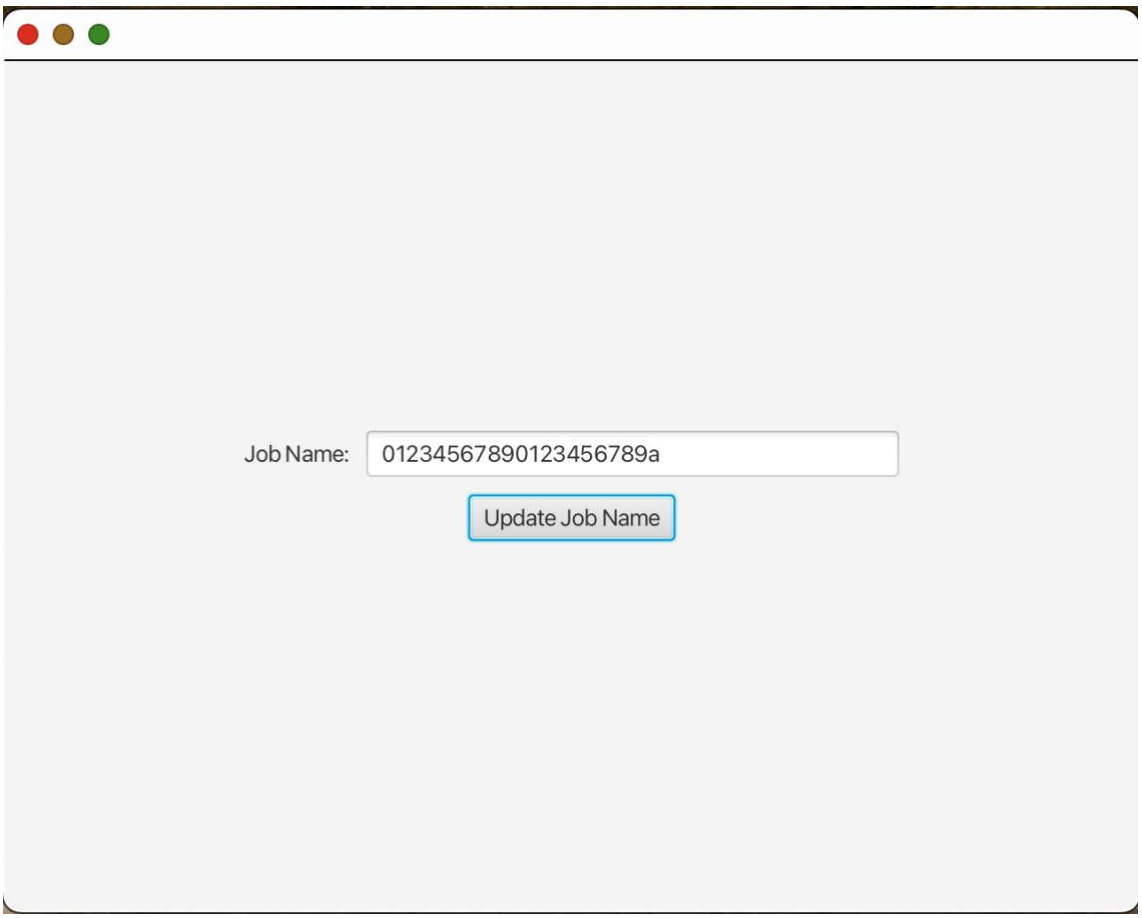
2. Test the functionality of updating the job name by pressing the Enter key while the text field is in focus.
3. Verify the functionality of the Update button in updating the job name.

```
jingjingyang@mac ex1 % java --module-path $PATH_TO_FX -
-add-modules javafx.controls TestJob.java
Updated Job:
        TestwithEnter: from 60 to 92 by 6 semitones
Updated Job:
        TestwithButton: from 60 to 92 by 6 semitones
```

4. Attempt to provide an empty string as the new job name and check the application's response.
5. Attempt to provide a job name that exceeds 20 charactersand check the application's response.

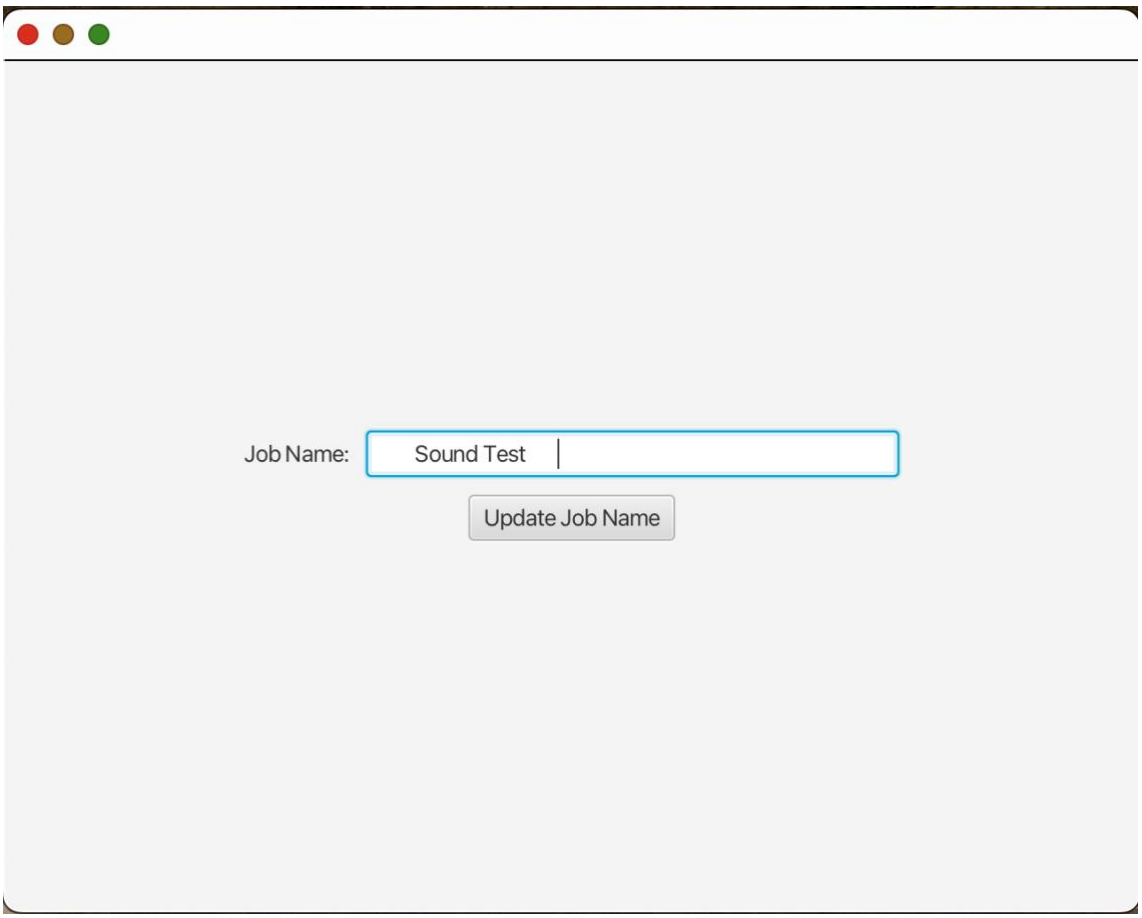
```
jingjingyang@mac ex1 % java --module-path $PATH_TO_FX -
-add-modules javafx.controls TestJob.java
Invalid Job Name. It should be 1-20 characters long.
Invalid Job Name. It should be 1-20 characters long.
```





6. Try to provide a job name that has leading and trailing spaces, and verify the application's response.

```
jingjingyang@mac ex1 % java --module-path $PATH_TO_FX -
-add-modules javafx.controls TestJob.java
Updated Job:
        Sound Test: from 60 to 92 by 6 semitones
Updated Job:
        Test123: from 60 to 92 by 6 semitones
```


Job Name:
Sound Test
Update Job Name

2 Exercise 2

2.1 Functionalities

- The JavaFX application was created with a GridPane as the layout for the scene.
- An instance of the Job class was added to the application and initialized with a job name and note values.
- The GridPane was configured to have two columns and four rows.
- Labels for "Job Name", "From Note", and "To Note" were added in the first column.
- Corresponding controls (a TextField for the job name and two spinners for the notes) were added in the second column.
- The TextField and the spinners were set to default values from the initial Job instance values.
- A button labeled "Update Job" was added to the GridPane.
- The button was set to span the two columns and was centered in the GridPane.
- An event handler was added to the button to collect the values from the TextField and spinners and use them to update the job instance when the button is clicked.
- When the job is updated, its details including job's name, starting note, ending note, and interval value are printed to the console.
- The application validates the job name entered by the user to ensure it's not empty, has no leading or trailing spaces, and is not longer than 20 characters.

2.2 Solution

- A JavaFX application was created with Application.launch().
- An instance of the Job class was added to the application and initialized with new Job().
- The GridPane configuration was achieved with new GridPane(), GridPane.setAlignment(), GridPane.setHgap(), GridPane.setVgap(), and GridPane.setPadding().
- Labels were added to the first column of the GridPane with GridPane.add().
- Corresponding controls (a TextField and two Spinners) were added to the second column of the GridPane with GridPane.add().
- The TextField and the spinners were set to default values with TextField.getText(), Spinner.getValue().
- A button labeled "Update Job" was added to the GridPane with GridPane.add().
- The button was set to span the two columns and centered in the GridPane with GridPane.setHalignment().
- An event handler was added to the button to collect the values from the TextField and spinners and use them to update the job instance when the button is clicked with Button.setOnAction().
- When the job is updated, its details are printed to the console with System.out.println().
- The Spinner constructor sets the "from note" and "to note" range from 0 to 127.

- The addListener method to fromNoteSpinner and toNoteSpinner values checks if "from note" is less than "to note", automatically disable the button if true using the setDisable(true) method.

2.3 TestJob.java

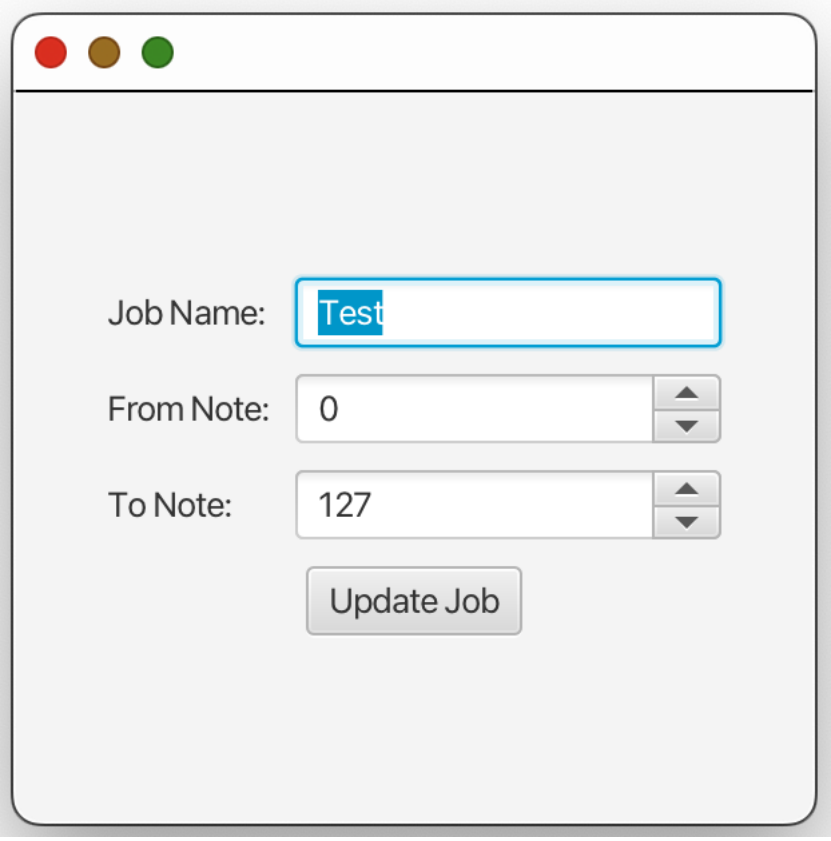
```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Spinner;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import javafx.geometry.HPos;

public class TestJob extends Application {
    private Job job;  // Job instance

    @Override
    public void start(Stage stage) {
        job = new Job("Test");  // Initialize a new Job
        job.setFromNote(0);
        job.setToNote(127); // Note must be 0...127

        GridPane grid = new GridPane();  // Create GridPane
        grid.setAlignment(Pos.CENTER);  // Center align the GridPane
        grid.setHgap(10);  // Set horizontal gap between grid cells
        grid.setVgap(10);  // Set vertical gap between grid cells
        grid.setPadding(new Insets(25));  // Set the padding for the GridPane to 25 units on all
sides

        Label jobNameLabel = new Label("Job Name:");  // Label for the job name field
        Label fromNoteLabel = new Label("From Note:");  // Label for the from note spinner
        Label toNoteLabel = new Label("To Note:");  // Label for the to note spinner
```

```
        grid.add(jobNameLabel, 0, 0);
        grid.add(fromNoteLabel, 0, 1);
        grid.add(toNoteLabel, 0, 2);

        TextField jobNameField = new TextField(job.getName());  // TextField for the job name
        Spinner<Integer> fromNoteSpinner = new Spinner<>(0, 127, job.getFromNote());  //
Spinner for the from note, with minimum value 0 and maximum value 127
        Spinner<Integer> toNoteSpinner = new Spinner<>(0, 127, job.getToNote());  // Spinner for
the to note, with minimum value 0 and maximum value 127
        grid.add(jobNameField, 1, 0);
        grid.add(fromNoteSpinner, 1, 1);
        grid.add(toNoteSpinner, 1, 2);

        Button btn = new Button("Update Job");  // Update button
        grid.add(btn, 0, 3, 2, 1);  // Adds the button to the grid at column 0, row 3, and makes it
span across 2 columns and 1 row.
        GridPane.setHalignment(btn, HPos.CENTER);

        // Add a listener to the 'fromNoteSpinner' value property
        fromNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
            // Set the button to disabled if the new value is greater than or equal to the
'toNoteSpinner' value
            btn.setDisable(newValue >= toNoteSpinner.getValue());
        });

        // Add a listener to the 'toNoteSpinner' value property
        toNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
            // Set the button to disabled if the new value is less than or equal to the
'fromNoteSpinner' value
            btn.setDisable(newValue <= fromNoteSpinner.getValue());
        });

        btn.setOnAction(e -> {  // Set action on button click
            String newName = jobNameField.getText().trim();  // Trim leading and trailing
whitespaces
```

```
            // Check if job name is not empty and not exceeding 20 characters
            if (newName.length() > 0 && newName.length() <= 20) {
                job.setName(newName);  // Set the new job name
                job.setFromNote(fromNoteSpinner.getValue());  // Update job from note
                job.setToNote(toNoteSpinner.getValue());  // Update job to note
                System.out.println("Updated Job: \n" + job);  // Print the updated job
            } else {
                System.out.println("Invalid Job Name. It should be 1-20 characters long.");  // Print
error if invalid
            }
        });

        Scene scene = new Scene(grid, 300, 275);  // Create Scene
        stage.setScene(scene);  // Set the scene
        stage.show();  // Show the stage
    }

    public static void main(String[] args) {
        launch();  // Launch the application
    }
}
```

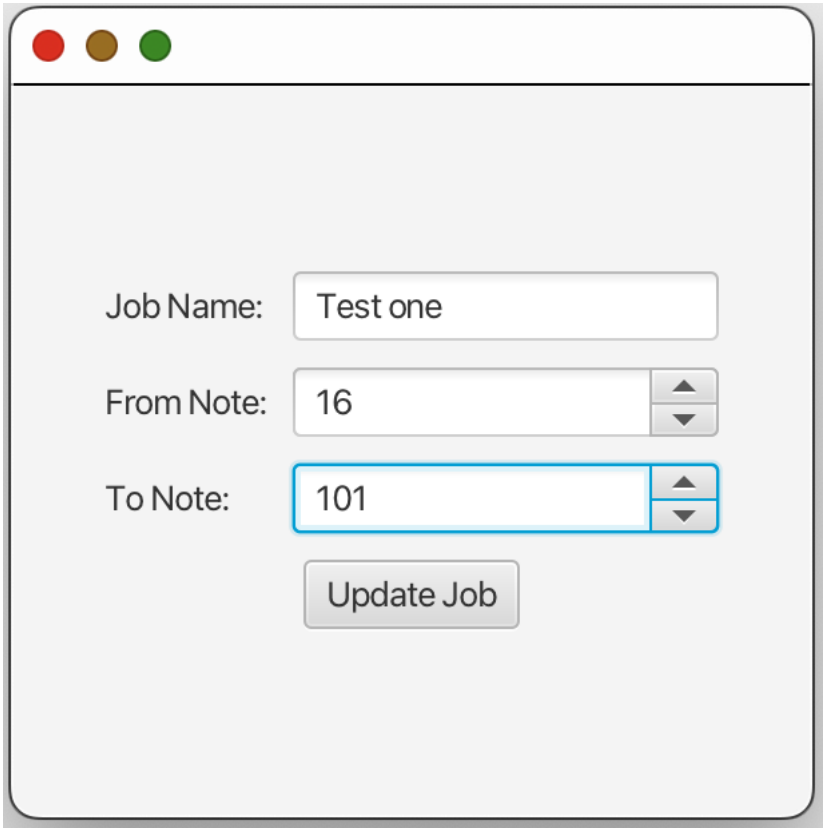
2.4 Testing

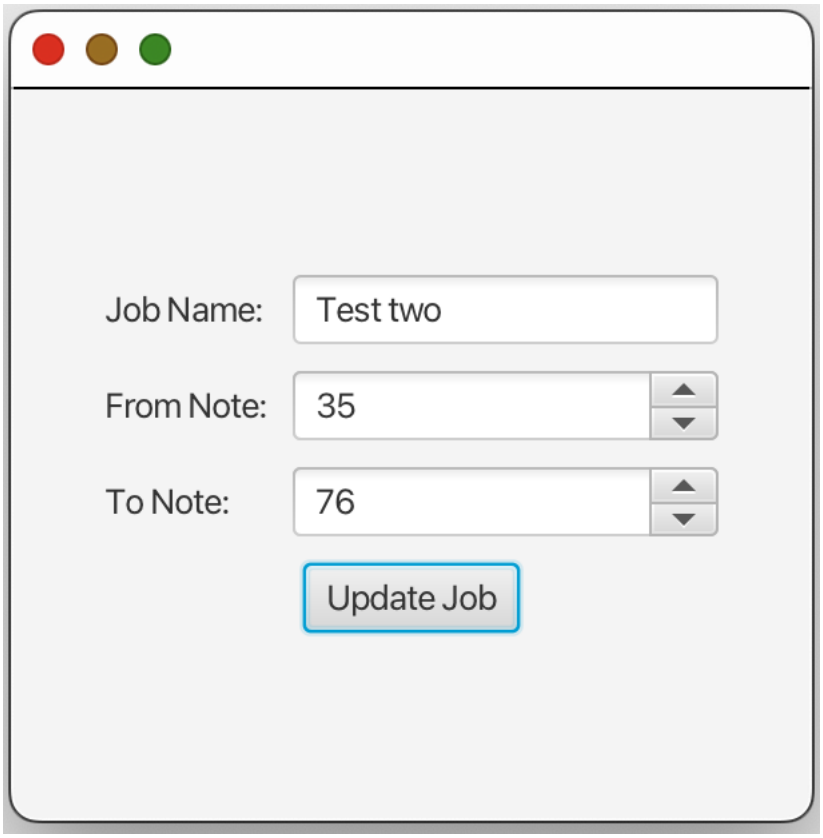
1. Check the initial display of the application.



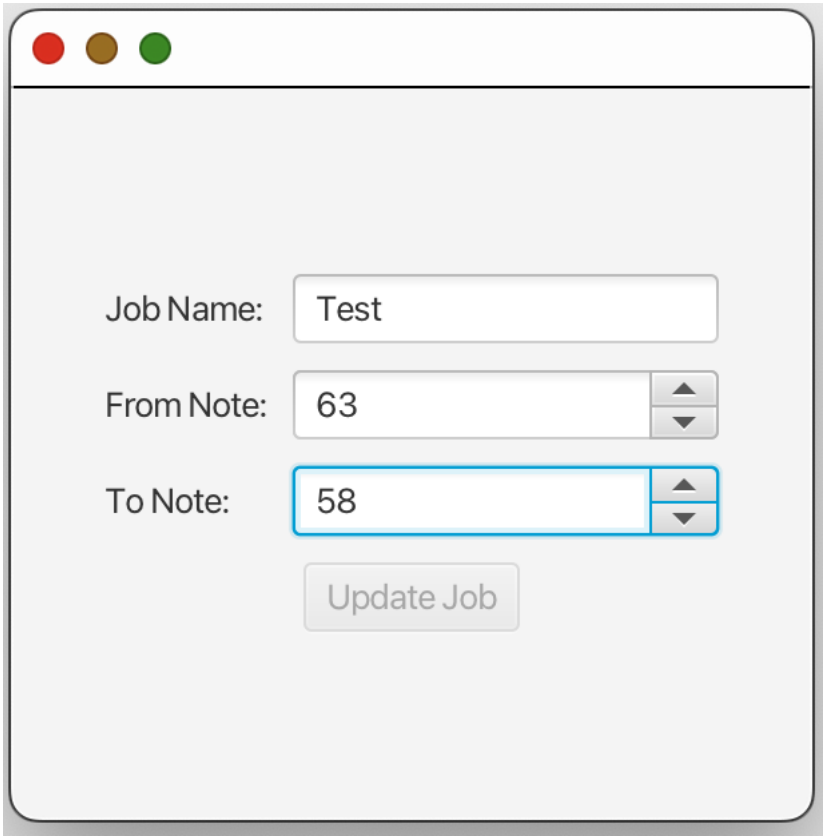
2. Test the functionality of updating the job name by clicking the Update button.

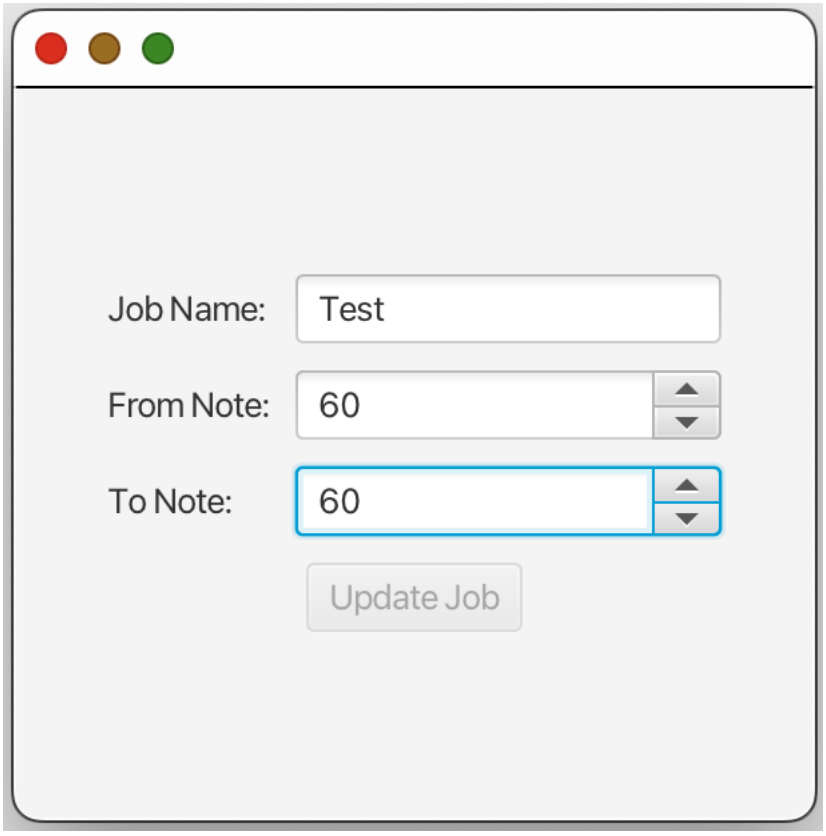
```
jingjingyang@mac ex2 % javac --module-path $PATH_TO_FX --add-modules
 javafx.controls TestJob.java
jingjingyang@mac ex2 % java --module-path $PATH_TO_FX --add-modules
javafx.controls TestJob.java
Updated Job:
Test: from 0 to 127 by 6 semitones
Updated Job:
Test one: from 16 to 101 by 6 semitones
Updated Job:
Test two: from 35 to 76 by 6 semitones
```





3. Verify the functionality of the Update Job button becoming disabled when the From Note value is not smaller than the To Note value.


Job Name:
Test
From Note:
63
To Note:
58
Update Job


Job Name:
Test
From Note:
60
To Note:
60
Update Job

2.5 Key Learning Points

- The **GridPane** layout in JavaFX dynamically creates rows and columns as components are added.
    - A component can be added to a specific location in the grid by specifying the column and row indices when adding the component.
    - When adding a component to a GridPane, its column span and row span can also be specified to have it span across multiple columns or rows.
- The GridPane's **setPadding** method is used to set the padding around the GridPane layout. The Insets object defines the size of the padding on all four sides of the component.
    - If the padding for all four sides is the same, the Insets(double topRightBottomLeft) constructor can be used to set the padding for all four sides to the same value.
- Spinner<Integer> is a shorthand way to create a Spinner object that will produce and accept Integer values, allowing you to set minimum, maximum, and initial values directly.
    - For more complex Spinner behavior, such as custom step sizes or value wrapping, a SpinnerValueFactory, like **IntegerSpinnerValueFactory** can be used. This allows for more control over the Spinner's behavior, but requires a bit more code.

3 Exercise 3

3.1 Functionalities

The application includes all the features as in Exercise 2 for updating the job name, from note, and to note, but adds a new feature to update the interval of the job.

The new feature is implemented by adding a group of radio buttons to the application, allowing users to choose the interval between notes in a job.

3.2 Solution

A ToggleGroup is created to group radio buttons, which ensures only one can be selected at a time. A list of Job.Interval enum values(ONE, THREE, SIX, TWELVE) is created and iterated to create radio buttons corresponding to each interval. These radio buttons are added to an HBox which is then placed inside a TitledPane, an expandable and collapsible container.

Each radio button is set with an ActionEvent handler, which upon clicking, calls the setInterval method on the Job instance to update the job's interval to the corresponding value. The updated job is then printed to the console.

The ToggleGroup is set to span the two columns in the 4th row in the GridPane, and the Update button is moved to the 5th row.

In addition, I set the window name by using the setTitle method on the Stage object.

3.3 Job.java v2

```
import java.util.UUID;

import java.util.List;

import java.util.ArrayList;



/**

 * Describes a sampling job with successive notes.

 */

public class Job {

    /**

     * Interval between notes in semitones.

     */

    public enum Interval {

        ONE(1),

        THREE(3),

        SIX(6),
```

```
    TWELVE(12);

    private final int value;

    private Interval(int value) {
        this.value = value;
    }

    /**
     * Gets the ordinal value of this enum instance.
     */
    public int getValue() {
        return value;
    }
}

// Internal count of instances created.
// Used to construct the default name for the job.
private static int count = 1;

public Job() {
    this("Job" + Job.count);
}

/**
 * Constructs a job with default values.
 */
public Job(String name) {
    this.name = name;
    this.id = UUID.randomUUID();
    this.fromNote = 40;
    this.toNote = 120;
    this.interval = Interval.SIX;

    Job.count++; // increase the internal counter
```

```
}

/**
 * Gets the name of the job.
 */
public String getName() {
    return this.name;
}

/**
 * Sets the name of the job.
 */
public void setName(String name) {
    this.name = name;
}

/**
 * Gets the unique identifier of the job.
 */
public UUID getId() {
    return this.id;
}

/**
 * Gets the first note of the range.
 */
public int getFromNote() {
    return this.fromNote;
}

/**
 * Sets the first note of the range.
 */
public void setFromNote(int note) {
    if (note < 0 || note > 127) {
```

```
            throw new IllegalArgumentException("Note must be 0...127");
        }
        this.fromNote = note;
    }

    /**
     * Gets the last note of the range.
     */
    public int getToNote() {
        return this.toNote;
    }

    /**
     * Sets the last note of the range.
     */
    public void setToNote(int note) {
        if (note < 0 || note > 127) {
            throw new IllegalArgumentException("Note must be 0...127");
        }
        this.toNote = note;
    }

    /**
     * Gets the interval between the notes.
     */
    public Interval getInterval() {
        return this.interval;
    }

    /**
     * Sets the interval between the notes.
     */
    public void setInterval(Interval i) {
        this.interval = i;
    }
```

```
/**
 * Gets the note duration.
 *
 * @return the note duration in milliseconds
 */
public int getNoteDuration() {
    return this.noteDuration;
}

/**
 * Sets the duration of each note.
 *
 * @param duration note duration (milliseconds)
 */
public void setNoteDuration(int duration) {
    if (duration <= 0) {
        throw new IllegalArgumentException("Note duration must be positive");
    }

    this.noteDuration = duration;
}

/**
 * Gets the note decay time.
 *
 * @return the note decay time in milliseconds
 */
public int getNoteDecay() {
    return this.noteDecay;
}

/**
 * Sets the decay of each note.
 *
```

```
     * @param decay note decay (milliseconds)
     */
    public void setNoteDecay(int decay) {
        if (decay <= 0) {
            throw new IllegalArgumentException("Note decay time must be positive");
        }

        this.noteDecay = decay;
    }

    /**
     * Gets the gap time between notes.
     *
     * @return the note gap time in milliseconds
     */
    public int getNoteGap() {
        return this.noteGap;
    }

    /**
     * Sets the gap time between notes.
     *
     * @param gap note gap time (milliseconds)
     */
    public void setNoteGap(int gap) {
        if (gap <= 0) {
            throw new IllegalArgumentException("Note gap time must be positive");
        }

        this.noteGap = gap;
    }

    public List<Integer> getNotes() {
        List<Integer> notes = new ArrayList<>();
```

```
    int note = this.fromNote;
    while (note <= this.toNote) {
        notes.add(note);
        note += this.interval.getValue();
    }

    return notes;
}

/**
 * Gets a string representation of the job.
 */
@Override
public String toString() {
    return String.format(
        "%s: from %d to %d by %d semitones, duration %d ms, decay %d ms, gap %d ms",
        this.getName(),
        this.getFromNote(),
        this.getToNote(),
        this.interval.getValue(),
        this.getNoteDuration(),
        this.getNoteDecay(),
        this.getNoteGap()
    );
}

//
// Private fields
//

private String name;
private UUID id;
private int fromNote;
private int toNote;
private Interval interval;
```

```
    private int noteDuration;  // milliseconds
    private int noteDecay;  // note decay time in ms
    private int noteGap;  // note gap time in ms
}
```

3.4 TestJob.java

```
import java.util.Arrays;
import java.util.List;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Spinner;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import javafx.geometry.HPos;

import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.control.TitledPane;
import javafx.scene.layout.HBox;

public class TestJob extends Application {
    private Job job;  // Job instance

    @Override
    public void start(Stage stage) {
        job = new Job("Test");  // Initialize a new Job
        job.setFromNote(40);
        job.setToNote(90); // Note must be 0...127
        //
        // GridPane
```

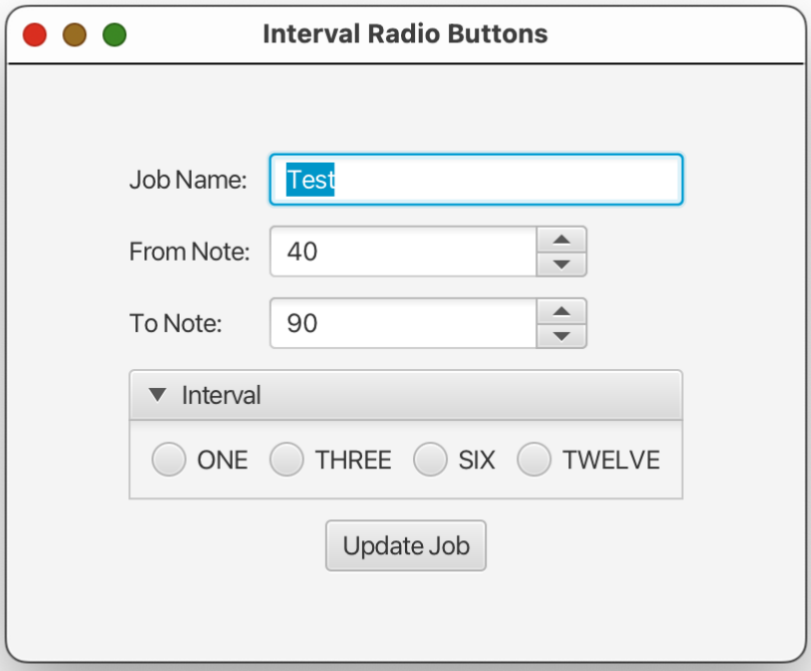
```
        //
        GridPane grid = new GridPane();  // Create GridPane
        grid.setAlignment(Pos.CENTER);  // Center align the GridPane
        grid.setHgap(10);  // Set horizontal gap between grid cells
        grid.setVgap(10);  // Set vertical gap between grid cells
        grid.setPadding(new Insets(25));  // Set the padding for the GridPane to 25 units on all sides
        //
        // Labels
        //
        Label jobNameLabel = new Label("Job Name:");  // Label for the job name field
        Label fromNoteLabel = new Label("From Note:");  // Label for the from note spinner
        Label toNoteLabel = new Label("To Note:");  // Label for the to note spinner
        grid.add(jobNameLabel, 0, 0);
        grid.add(fromNoteLabel, 0, 1);
        grid.add(toNoteLabel, 0, 2);
        //
        // TextField and Spinner
        //
        TextField jobNameField = new TextField(job.getName());  // TextField for the job name
        Spinner<Integer> fromNoteSpinner = new Spinner<>(0, 127, job.getFromNote());  // Spinner for the from note, with
minimum value 0 and maximum value 127
        Spinner<Integer> toNoteSpinner = new Spinner<>(0, 127, job.getToNote());  // Spinner for the to note, with
minimum value 0 and maximum value 127
        grid.add(jobNameField, 1, 0);
        grid.add(fromNoteSpinner, 1, 1);
        grid.add(toNoteSpinner, 1, 2);
        //
        // ToggleGroup
        //
        ToggleGroup group = new ToggleGroup();
        List<Job.Interval> intervals = Arrays.asList(Job.Interval.ONE, Job.Interval.THREE, Job.Interval.SIX,
Job.Interval.TWELVE);
        HBox hbox = new HBox();
        hbox.setSpacing(10); // Sets the space to 10
```

```
        for (Job.Interval interval : intervals) {
            RadioButton button = new RadioButton(interval.name());
            button.setToggleGroup(group);
            button.setOnAction(event -> {
                job.setInterval(interval);
                System.out.println(job);
            });
            hbox.getChildren().add(button);
        }

        TitledPane tp = new TitledPane("Interval", hbox);
        tp.setCollapsible(true);
        grid.add(tp, 0, 3, 2, 1);  // Adds the TitlePane to the grid at column 0, row 3, and makes it span across 2 columns
and 1 row.
        //
        // Button
        //
        Button btn = new Button("Update Job");  // Update button
        grid.add(btn, 0, 4, 2, 1);  // Adds the button to the grid at column 0, row 4, and makes it span across 2 columns and
1 row.
        GridPane.setHalignment(btn, HPos.CENTER);

        // Add a listener to the 'fromNoteSpinner' value property
        fromNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
            // Set the button to disabled if the new value is greater than or equal to the 'toNoteSpinner' value
            btn.setDisable(newValue >= toNoteSpinner.getValue());
        });

        // Add a listener to the 'toNoteSpinner' value property
        toNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
            // Set the button to disabled if the new value is less than or equal to the 'fromNoteSpinner' value
            btn.setDisable(newValue <= fromNoteSpinner.getValue());
        });

        btn.setOnAction(e -> {  // Set action on button click
```

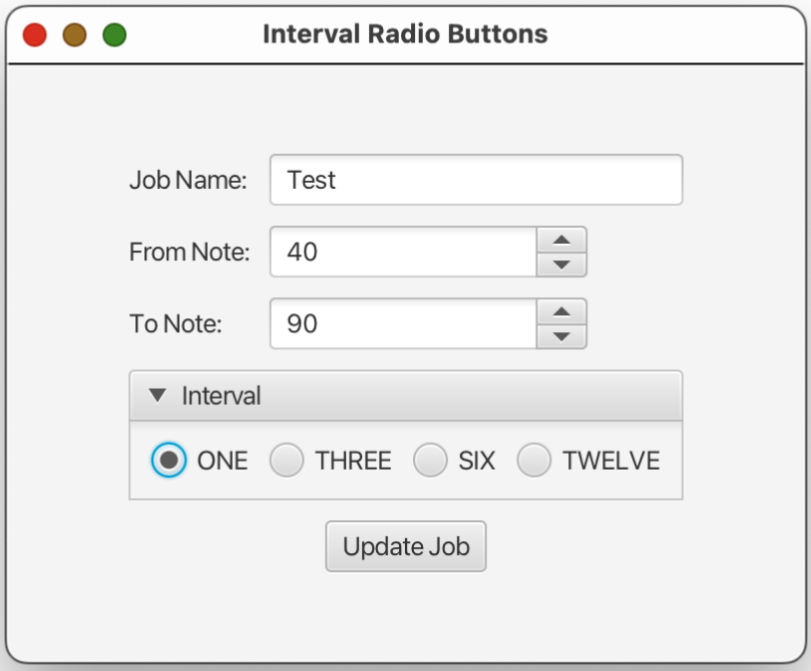
```
        String newName = jobNameField.getText().trim();  // Trim leading and trailing whitespaces


        // Check if job name is not empty and not exceeding 20 characters
        if (newName.length() > 0 && newName.length() <= 20) {
            job.setName(newName);  // Set the new job name
            job.setFromNote(fromNoteSpinner.getValue());  // Update job from note
            job.setToNote(toNoteSpinner.getValue());  // Update job to note
            System.out.println("Updated Job: \n" + job);  // Print the updated job
        } else {
            System.out.println("Invalid Job Name. It should be 1-20 characters long.");  // Print error if invalid
        }
    });
    //
    // Scene
    //
    Scene scene = new Scene(grid, 400, 300);
    stage.setTitle("Interval Radio Buttons"); // Create Scene
    stage.setScene(scene);  // Set the scene
    stage.show();  // Show the stage
  }


  public static void main(String[] args) {
    launch();  // Launch the application
  }
}
```

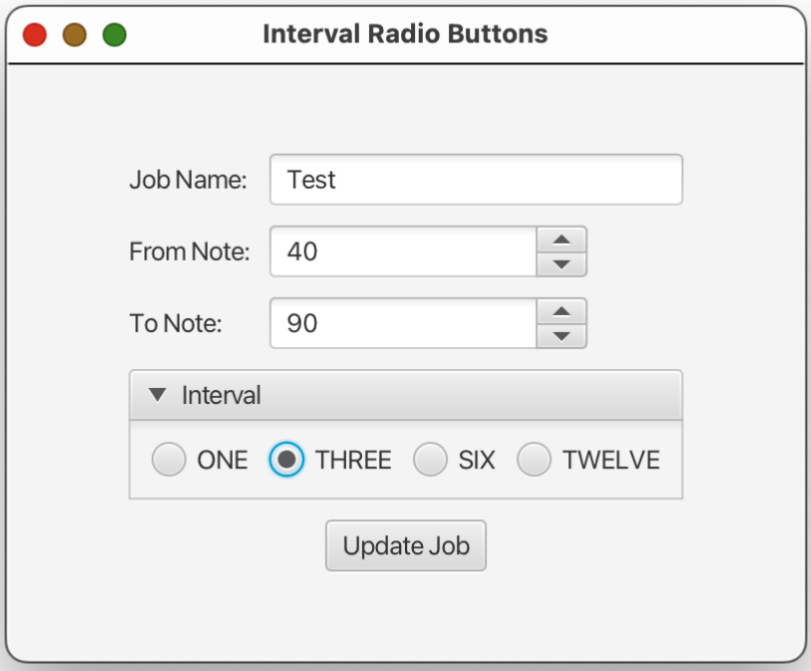
3.5 Testing

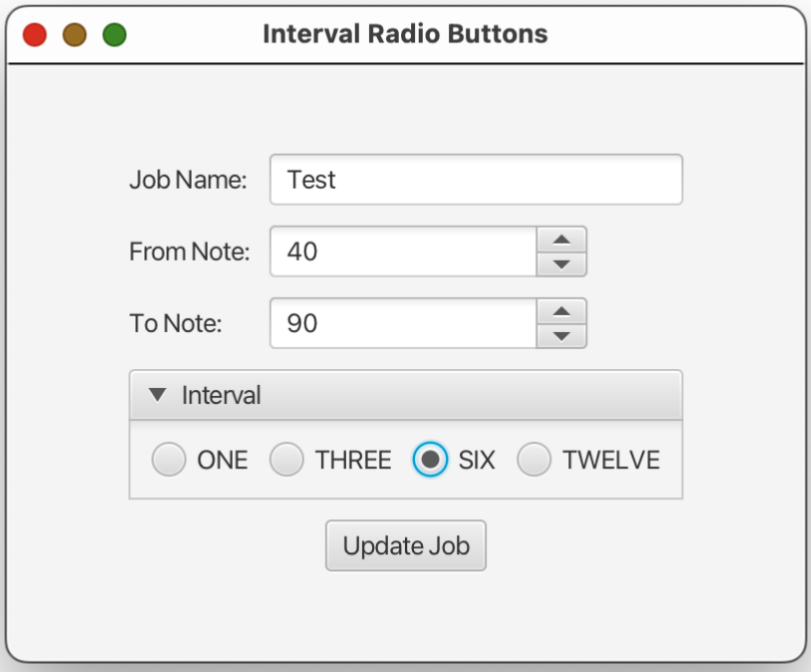
1. Check the initial display of the application.

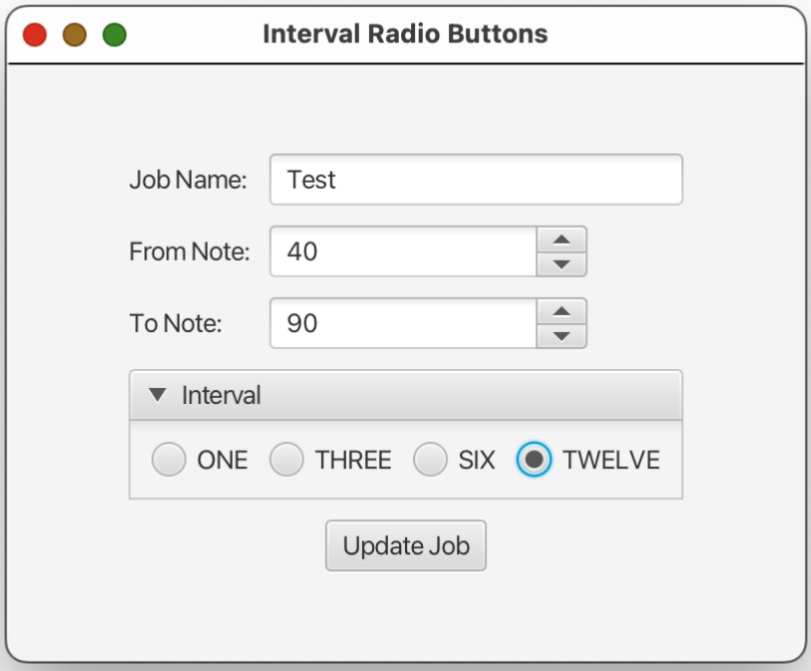


2. Check the selection of the interval, one a time.




Interval Radio Buttons
Job Name:
Test
From Note:
40
To Note:
90
Interval
ONE
THREE
SIX
TWELVE
Update Job


Interval Radio Buttons
Job Name:
Test
From Note:
40
To Note:
90
Interval
ONE
THREE
SIX
TWELVE
Update Job



3. Check the output when clicking any radio button.

```
jingjingyang@mac ex3 % javac --module-path $PATH_TO_FX --add-modules javafx
.controls TestJob.java
jingjingyang@mac ex3 % java --module-path $PATH_TO_FX --add-modules javafx.
controls TestJob.java
Test: from 40 to 90 by 1 semitones, duration 0 ms, decay 0 ms, gap 0 ms
Test: from 40 to 90 by 3 semitones, duration 0 ms, decay 0 ms, gap 0 ms
Test: from 40 to 90 by 6 semitones, duration 0 ms, decay 0 ms, gap 0 ms
Test: from 40 to 90 by 12 semitones, duration 0 ms, decay 0 ms, gap 0 ms
```

4 Exercise 4

4.1 Functionalities

The application includes all the features as in Exercise 3, but adds some new features as below:

1. Three sliders with associated labels for the note duration, note decay, and gap between notes.
2. A checkbox for default times.
3. A total time label.
4. Graphical representation of the total note time on a Canvas.
5. An update button to update the note timing values in the job.

4.2 Solution

1. Sliders: Three Slider components are created with minimum value of 100ms and different maximum values for note duration, note decay, and gap. The Slider values are connected to Labels to display the current value and also set to update the job's note timings when changed.
2. Checkbox: A checkbox is added which toggles 'Use default times'. If the checkbox is selected, the sliders are disabled and the job's note timings are set to default values. If unchecked, the sliders are enabled and the job's note timings are set according to the sliders' value.
3. Total Time Label: A label is added to display the total note time, which is updated whenever the note timings are changed.
4. Canvas: A Canvas component is added to display a graphical representation of the total note time. This is achieved by drawing rectangular blocks inside the Canvas where their widths are in correct proportions to the note timing. The Canvas is updated whenever the note timings are changed.
5. Update Button: An 'Update Job' button is added which updates the job when clicked. The button is disabled if the 'fromNoteSpinner' value is greater than or equal to the 'toNoteSpinner' value, and vice versa. The updated job is printed on the console when the button is clicked.

4.3 Job.java v3

```
import java.util.UUID;

import java.util.List;

import java.util.ArrayList;



/**

 * Describes a sampling job with successive notes.

 */

public class Job {

    /**

     * Interval between notes in semitones.

     */
```

```
public enum Interval {
    ONE(1),
    THREE(3),
    SIX(6),
    TWELVE(12);

    private final int value;

    private Interval(int value) {
        this.value = value;
    }

    /**
     * Gets the ordinal value of this enum instance.
     */
    public int getValue() {
        return value;
    }
}

// Internal count of instances created.
// Used to construct the default name for the job.
private static int count = 1;

public Job() {
    this("Job" + Job.count);
}

/**
 * Constructs a job with default values.
 */
public Job(String name) {
    this.name = name;
    this.id = UUID.randomUUID();
    this.fromNote = 40;
```

```
        this.toNote = 120;
        this.interval = Interval.SIX;

        // All note duration values are in milliseconds:
        this.noteDuration = 1000;
        this.noteDecay = 500;
        this.noteGap = 100;

        Job.count++; // increase the internal counter
    }

    /**
     * Sets the name of the job.
     *
     * @param name the new name
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Gets the name of the job.
     */
    public String getName() {
        return this.name;
    }

    /**
     * Gets the unique identifier of the job.
     */
    public UUID getId() {
        return this.id;
    }

    /**
```

```
    * Gets the first note of the range.
    */
    public int getFromNote() {
        return this.fromNote;
    }

    /**
    * Sets the first note of the range.
    */
    public void setFromNote(int note) {
        if (note < 0 || note > 127) {
            throw new IllegalArgumentException("Note must be 0...127");
        }
        this.fromNote = note;
    }

    /**
    * Gets the last note of the range.
    */
    public int getToNote() {
        return this.toNote;
    }

    /**
    * Sets the last note of the range.
    */
    public void setToNote(int note) {
        if (note < 0 || note > 127) {
            throw new IllegalArgumentException("Note must be 0...127");
        }
        this.toNote = note;
    }

    /**
    * Gets the interval between the notes.
```

```
     */
    public Interval getInterval() {
        return this.interval;
    }

    /**
     * Sets the interval between the notes.
     */
    public void setInterval(Interval i) {
        this.interval = i;
    }

    /**
     * Gets the note duration.
     *
     * @return the note duration in milliseconds
     */
    public int getNoteDuration() {
        return this.noteDuration;
    }

    /**
     * Sets the duration of each note.
     *
     * @param duration note duration (milliseconds)
     */
    public void setNoteDuration(int duration) {
        if (duration <= 0) {
            throw new IllegalArgumentException("Note duration must be positive");
        }

        this.noteDuration = duration;
    }

    /**
```

```
    * Gets the note decay time.
    *
    * @return the note decay time in milliseconds
    */
    public int getNoteDecay() {
        return this.noteDecay;
    }

    /**
    * Sets the decay of each note.
    *
    * @param decay note decay (milliseconds)
    */
    public void setNoteDecay(int decay) {
        if (decay <= 0) {
            throw new IllegalArgumentException("Note decay time must be positive");
        }

        this.noteDecay = decay;
    }

    /**
    * Gets the gap time between notes.
    *
    * @return the note gap time in milliseconds
    */
    public int getNoteGap() {
        return this.noteGap;
    }

    /**
    * Sets the gap time between notes.
    *
    * @param gap note gap time (milliseconds)
    */
```

```
public void setNoteGap(int gap) {
    if (gap <= 0) {
        throw new IllegalArgumentException("Note gap time must be positive");
    }

    this.noteGap = gap;
}

public List<Integer> getNotes() {
    List<Integer> notes = new ArrayList<>();

    int note = this.fromNote;
    while (note <= this.toNote) {
        notes.add(note);
        note += this.interval.getValue();
    }

    return notes;
}

/**
 * Gets a string representation of the job.
 */
@Override
public String toString() {
    return String.format(
        "%s: from %d to %d by %d semitones, duration %d ms, decay %d ms, gap %d ms",
        this.getName(),
        this.getFromNote(),
        this.getToNote(),
        this.interval.getValue(),
        this.getNoteDuration(),
        this.getNoteDecay(),
        this.getNoteGap()
    );
```

```
    }

    //
    // Private fields
    //

    private String name;
    private UUID id;
    private int fromNote;
    private int toNote;
    private Interval interval;
    private int noteDuration;  // milliseconds
    private int noteDecay;  // note decay time in ms
    private int noteGap;  // note gap time in ms
}
```

4.4 TestJob.java

```
import java.util.Arrays;
import java.util.List;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Spinner;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import javafx.geometry.HPos;

import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.control.TitledPane;
import javafx.scene.layout.HBox;
```

```
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.control.Slider;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.VBox;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.paint.CycleMethod;

public class TestJob extends Application {
    private Job job;  // Job instance

    @Override
    public void start(Stage stage) {
        job = new Job("Test"); // Initialize job instance

        //
        // GridPane
        //
        GridPane grid = new GridPane();  // Create GridPane
        grid.setAlignment(Pos.CENTER);  // Center align the GridPane
        grid.setHgap(10);  // Set horizontal gap between grid cells
        grid.setVgap(15);  // Set vertical gap between grid cells
        grid.setPadding(new Insets(25));  // Set the padding for the GridPane to 25 units on all sides

        //
        // Text
        //
        Text heading = new Text("Please customize the job details");
        heading.setFont(new Font(20)); // Set font size to 20
```

```
        grid.add(heading, 0, 0, 2, 1);  // Adds the heading to the grid at column 0, row 0, and makes it span across 2
columns and 1 row.
        GridPane.setHalignment(heading, HPos.CENTER);

        //
        // Labels
        //
        Label jobNameLabel = new Label("Job Name:");  // Label for the job name field
        Label fromNoteLabel = new Label("From Note:");  // Label for the from note spinner
        Label toNoteLabel = new Label("To Note:");  // Label for the to note spinner
        grid.add(jobNameLabel, 0, 1);
        grid.add(fromNoteLabel, 0, 2);
        grid.add(toNoteLabel, 0, 3);

        //
        // TextField and Spinners
        //
        TextField jobNameField = new TextField(job.getName());  // TextField for the job name
        Spinner<Integer> fromNoteSpinner = new Spinner<>(0, 127, job.getFromNote());  // Spinner for the from note, with
minimum value 0 and maximum value 127
        Spinner<Integer> toNoteSpinner = new Spinner<>(0, 127, job.getToNote());  // Spinner for the to note, with
minimum value 0 and maximum value 127
        grid.add(jobNameField, 1, 1);
        grid.add(fromNoteSpinner, 1, 2);
        grid.add(toNoteSpinner, 1, 3);

        //
        // ToggleGroup
        //
        ToggleGroup group = new ToggleGroup();
        List<Job.Interval> intervals = Arrays.asList(Job.Interval.ONE, Job.Interval.THREE, Job.Interval.SIX,
Job.Interval.TWELVE);
        HBox hbox = new HBox();
        hbox.setSpacing(10); // Sets the space to 10
```

```
        for (Job.Interval interval : intervals) {
            RadioButton button = new RadioButton(interval.name());
            button.setToggleGroup(group);

            // Check if interval is default value
            if (interval == job.getInterval()) {
                // If so, set this radio button as selected
                button.setSelected(true);
            }

            button.setOnAction(event -> {
                job.setInterval(interval);
            });
            hbox.getChildren().add(button);
        }

        TitledPane tp = new TitledPane("Interval", hbox);
        tp.setCollapsible(true);
        grid.add(tp, 0, 4, 2, 1);  // Adds the TitledPane to the grid at column 0, row 4, and makes it span across 2 columns
and 1 row.

        //
        // Sliders
        //
        Label durationLabel = new Label("Duration: " + job.getNoteDuration() + " ms");
        Label decayLabel = new Label("Decay: " + job.getNoteDecay() + " ms");
        Label gapLabel = new Label("Gap: " + job.getNoteGap() + " ms");

        Slider durationSlider = new Slider(100, 5000, job.getNoteDuration());// note duration min 100ms, max 5000ms
        Slider decaySlider = new Slider(100, 4500, job.getNoteDecay());// note decay min 100ms, max 4500ms
        Slider gapSlider = new Slider(100, 500, job.getNoteGap());// note gap min 100ms, max 500ms

        // set slider properties
        durationSlider.setShowTickMarks(true);// display tick marks on the slider
        decaySlider.setShowTickMarks(true);
```

```
        gapSlider.setShowTickMarks(true);
        durationSlider.setShowTickLabels(true);// show labels for the tick marks
        decaySlider.setShowTickLabels(true);
        gapSlider.setShowTickLabels(true);
        durationSlider.setMajorTickUnit(100);// set the unit distance between major tick marks
        decaySlider.setMajorTickUnit(100);
        gapSlider.setMajorTickUnit(10);
        durationSlider.setBlockIncrement(100);// thumb move when using arrow keys
        decaySlider.setBlockIncrement(100);
        gapSlider.setBlockIncrement(10);

        VBox vbox = new VBox();
        vbox.setSpacing(10); // Sets the space to 10
        vbox.getChildren().addAll(durationLabel, durationSlider, decayLabel, decaySlider, gapLabel, gapSlider);

        TitledPane tp2 = new TitledPane("Note Times", vbox);
        tp2.setCollapsible(true);
        grid.add(tp2, 0, 5, 2, 1);  // Adds the TitledPane to the grid at column 0, row 5, and makes it span across 2 columns
and 1 row.

        //
        // CheckBox
        //
        CheckBox checkBox = new CheckBox("Use default times");
        grid.add(checkBox, 0, 6, 2, 1);  // Adds the checkBox to the grid at column 0, row 6, and makes it span across 2
columns and 1 row.

        //
        // Label to show total note time
        //
        Label totaltimeLabel=new Label("Total Note Time: " + job.getNoteDuration() + job.getNoteDecay() +
job.getNoteGap() + " ms");
        grid.add(totaltimeLabel, 0, 7, 2, 1);  // Adds the time label to the grid at column 0, row 7, and makes it span across
2 columns and 1 row.
```

```
        //
        // Canvas
        //
        Canvas canvas = new Canvas(300, 50);
        drawRectangles(canvas, totaltimeLabel); // initializes the canvas
        grid.add(canvas, 0, 8, 2, 1);  // Adds the canvas to the grid at column 0, row 8, and makes it span across 2 columns
and 1 row.


        //
        // Button
        //
        Button btn = new Button("Update Job");  // Update button
        grid.add(btn, 0, 9, 2, 1);  // Adds the button to the grid at column 0, row 9, and makes it span across 2 columns and
1 row.
        GridPane.setHalignment(btn, HPos.CENTER);


        btn.setOnAction(e -> {  // Set action on button click
            String newName = jobNameField.getText().trim();  // Trim leading and trailing whitespaces


            // Check if job name is not empty and not exceeding 20 characters
            if (newName.length() > 0 && newName.length() <= 20) {
                job.setName(newName);  // Set the new job name
                job.setFromNote(fromNoteSpinner.getValue());  // Update job from note
                job.setToNote(toNoteSpinner.getValue());  // Update job to note
                System.out.println("Updated Job: \n" + job);  // Print the updated job
            } else {
                System.out.println("Invalid Job Name. It should be 1-20 characters long.");  // Print error if invalid
            }
        });


        //
        // Listeners
        //
        // Add a listener to the 'fromNoteSpinner' value property
        fromNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
```

```
        // Set the button to disabled if the new value is greater than or equal to the 'toNoteSpinner' value
        btn.setDisable(newValue >= toNoteSpinner.getValue());
        // update job
        job.setFromNote(newValue.intValue());
    });

    // Add a listener to the 'toNoteSpinner' value property
    toNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
        // Set the button to disabled if the new value is less than or equal to the 'fromNoteSpinner' value
        btn.setDisable(newValue <= fromNoteSpinner.getValue());
        // update job
        job.setToNote(newValue.intValue());
    });

    // Add listeners to the sliders value property
    durationSlider.valueProperty().addListener((observable, oldValue, newValue) -> {
        durationLabel.setText("Duration: " + newValue.intValue() + " ms");
        // update job
        job.setNoteDuration(newValue.intValue());
        drawRectangles(canvas, totaltimeLabel); // update the canvas
    });
    decaySlider.valueProperty().addListener((observable, oldValue, newValue) -> {
        decayLabel.setText("Decay: " + newValue.intValue() + " ms");
        // update job
        job.setNoteDecay(newValue.intValue());
        drawRectangles(canvas, totaltimeLabel); // update the canvas
    });
    gapSlider.valueProperty().addListener((observable, oldValue, newValue) -> {
        gapLabel.setText("Gap: " + newValue.intValue() + " ms");
        // update job
        job.setNoteGap(newValue.intValue());
        drawRectangles(canvas, totaltimeLabel); // update the canvas
    });

    // Add a listener to the checkBox value property
```

```
    checkBox.selectedProperty().addListener((observable, oldValue, newValue) -> {
        durationSlider.setDisable(newValue);
        decaySlider.setDisable(newValue);
        gapSlider.setDisable(newValue);

        if (newValue) {
            job.setNoteDuration(1000);// default duration 1000ms
            job.setNoteDecay(500);// default decay 500ms
            job.setNoteGap(100);// default gap 100ms
        } else {
            // Use the latest values if newValue is false
            job.setNoteDuration((int) durationSlider.getValue());
            job.setNoteDecay((int) decaySlider.getValue());
            job.setNoteGap((int) gapSlider.getValue());
        }

        drawRectangles(canvas, totaltimeLabel); // update the canvas
    });

    //
    // Scene
    //
    Scene scene = new Scene(grid, 400, 750);
    stage.setTitle("Note Times Slider and Canvas"); // Create Scene
    stage.setScene(scene);  // Set the scene
    stage.show();  // Show the stage
}

private void drawRectangles(Canvas canvas, Label totaltimeLabel) {
    GraphicsContext gc = canvas.getGraphicsContext2D();

    // Calculate the total
    int total = job.getNoteDuration() + job.getNoteDecay() + job.getNoteGap();
    totaltimeLabel.setText("Total Note Time: " + total + " ms");
```

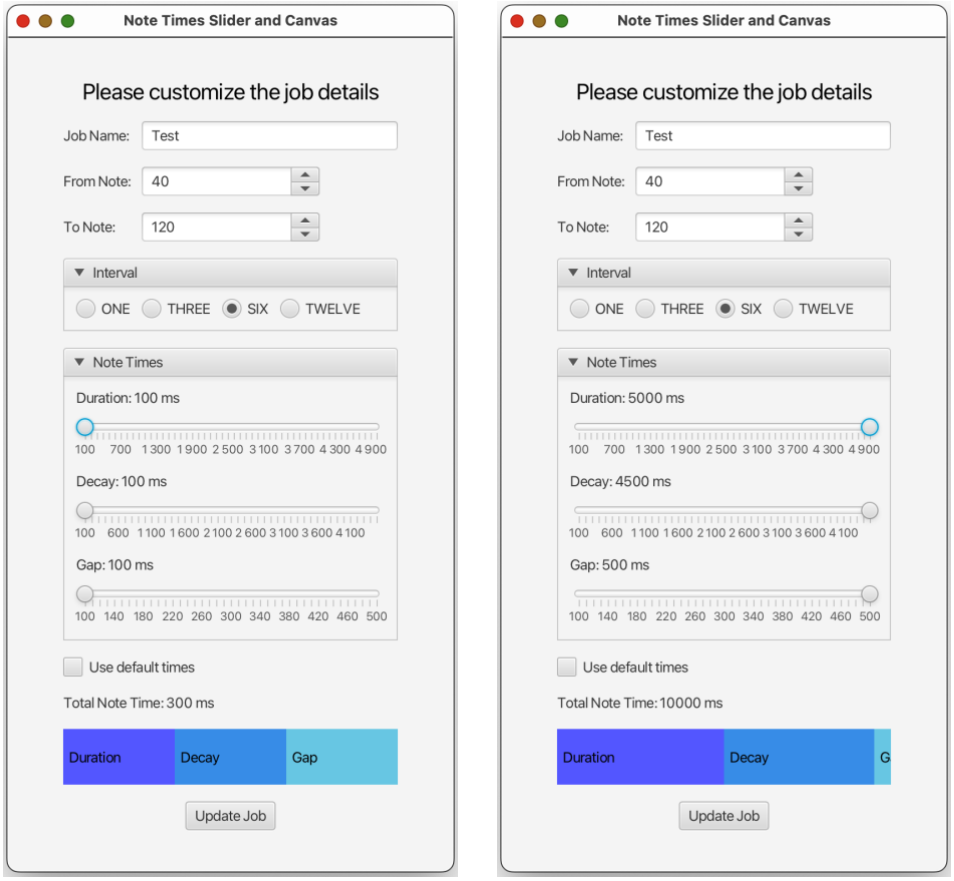
```
        // Calculate the percentage of each attribute
        double durationPercentage = (double) job.getNoteDuration() / total;
        double decayPercentage = (double) job.getNoteDecay() / total;
        double gapPercentage = (double) job.getNoteGap() / total;


        //
        // Option 1
        //
        // Create stops for the gradient
        Stop[] stops = new Stop[] {
            new Stop(0, Color.rgb(83, 86, 255)),
            new Stop(durationPercentage, Color.rgb(83, 86, 255)),
            new Stop(durationPercentage, Color.rgb(55, 140, 231)),
            new Stop(durationPercentage + decayPercentage, Color.rgb(55, 140, 231)),
            new Stop(durationPercentage + decayPercentage, Color.rgb(103, 198, 227)),
            new Stop(1, Color.rgb(103, 198, 227))
        };
        // Create a linear gradient
        LinearGradient gradient = new LinearGradient(0, 0, 1, 0, true, CycleMethod.NO_CYCLE, stops);
        // Draw the gradient across the entire canvas
        gc.setFill(gradient);
        gc.fillRect(0, 0, canvas.getWidth(), 50);
        // End of option 1


        // //
        // // Option 2
        // //
        // // Redraw the rectangles using percentages
        // gc.setFill(Color.rgb(142, 122, 181));
        // gc.fillRect(0, 0, durationPercentage * canvas.getWidth(), 50);
        // gc.setFill(Color.rgb(183, 132, 183));
        // gc.fillRect(durationPercentage * canvas.getWidth(), 0, decayPercentage * canvas.getWidth(), 50);
        // gc.setFill(Color.rgb(228, 147, 179));
        // gc.fillRect((durationPercentage + decayPercentage) * canvas.getWidth(), 0, gapPercentage * canvas.getWidth(),
50);
```

```
        // // End of option 2


        // Set the fill color for the text
        gc.setFill(Color.BLACK);


        // Add text to the rectangles
        gc.fillText("Duration", 5, 30);
        gc.fillText("Decay", durationPercentage * canvas.getWidth() + 5, 30);
        gc.fillText("Gap", (durationPercentage + decayPercentage) * canvas.getWidth() + 5, 30);
    }



    public static void main(String[] args) {
        launch();  // Launch the application
    }
}
```

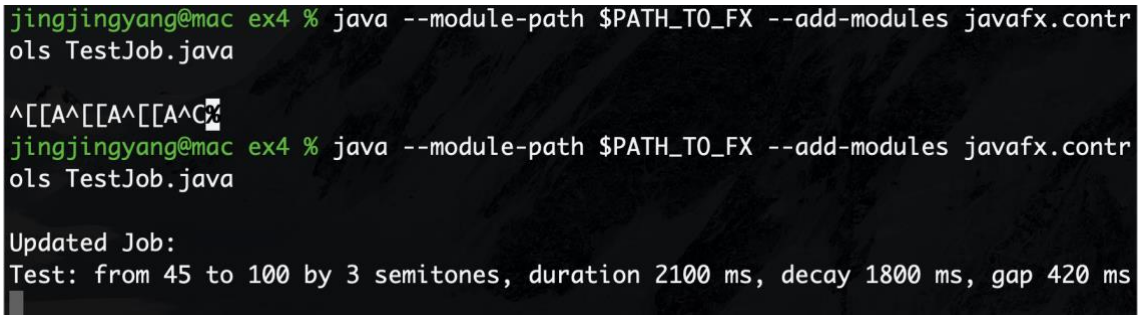
4.5 Testing

1. Check the initial display of the application.

2. Check the range of sliders.



3. Check functionalities of the update button.



4. Check functionalities of the checkbox, which uses default times when checked.

5 Exercise 5

5.1 Pre-knowledge

A generic type is a class or interface that is parameterized over types. This means it can operate on objects of various types while providing compile-time type safety. The type(s) operated on by a generic class or method are specified in angle brackets. For example, in Java, the List<E> interface is a generic type, where E is the type of elements in the list. You can create a list of integers with List<Integer>, a list of strings with List<String>, and so on.

JavaBeans naming convention is a standard in Java for naming methods, particularly getter and setter methods for properties. According to the convention:

- A property name of type Type would have methods:
  - Type getName(): a getter method to get the value of name.
  - void setName(Type name): a setter method to set the value of name.
- For boolean properties, the getter method can alternatively be named isName().

For example, if a class has a property color of type String, the getter method would be String getColor() and the setter method would be void setColor(String color).

5.2 Functionalities

- The application represents a musical job that consists of a series of notes.
- Each note has a number, velocity, start time, and end time.
- The application displays the notes in a well-organized TableView, allowing users to easily view and understand the various properties of each note within the job.
- For each note (n), there are a number of different velocities (m) that the note can be played at. This creates a combination of note and velocity pairs, resulting in n * m total notes.

5.3 Solution

- The application generates an ArrayList of Note objects, each representing a unique note in the job.
- The ArrayList of Note objects is converted into an ObservableList which is used as the data source for the TableView.
- The TableView includes four columns: Note, Velocity, Start Time (in ms), and End Time (in ms).

5.4 Time Calculation

The application calculates the start and end times for each note considering the note's duration, decay, and gap.

1. **Start Time**: The start time for each note is determined by the current time in the note sequence. The current time is initially set to 0 for the first

note, and for each subsequent note, it's set to the end time of the previous note plus the gap.

2. **End Time**: The end time for each note is calculated by adding the note's duration and decay to its start time.

The calculation is done in the following way:

```
// Set the start time of the note to the current time
note.setStartTime(currentTime);
// Set the end time of the note to the start time plus note duration and decay
note.setEndTime(currentTime + noteDuration + noteDecay);
```

Then, the current time is updated for the next note by adding the note duration, decay, and gap:

```
// Update the current time for the next note by adding note duration, decay and gap
currentTime += noteDuration + noteDecay + noteGap;
```

This method of calculation ensures that:

- Each note starts playing after the previous note has finished, including its decay time and the gap.
- The duration of a note does not include the decay time or the gap. The duration is purely the time for which the note is played at its peak.
- The decay time is the time taken for the note to fade away after its duration has passed. The decay starts immediately after the note's duration ends.
- The gap is the silent time between the end of one note (including its decay) and the start of the next note.

5.5 Job.java v4

```
import java.util.UUID;

import java.util.List;

import java.util.ArrayList;

import java.util.Collections;


/**

 * Describes a sampling job with successive notes.

 */

public class Job {

    /**

     * Interval between notes in semitones.

     */

    public enum Interval {

        ONE(1),

        THREE(3),

        SIX(6),

        TWELVE(12);
```

```
    private final int value;

    private Interval(int value) {
        this.value = value;
    }

    /**
     * Gets the ordinal value of this enum instance.
     */
    public int getValue() {
        return value;
    }
}

// Internal count of instances created.
// Used to construct the default name for the job.
private static int count = 1;

public Job() {
    this("Job" + Job.count);
}

/**
 * Constructs a job with default values.
 */
public Job(String name) {
    this.name = name;
    this.id = UUID.randomUUID();
    this.fromNote = 40;
    this.toNote = 120;
    this.interval = Interval.SIX;

    // All note duration values are in milliseconds:
    this.noteDuration = 1000;
```

```
    this.noteDecay = 500;
    this.noteGap = 100;

    this.velocities = new ArrayList<Integer>();
    this.velocities.add(90);

    Job.count++; // increase the internal counter
}

/**
 * Sets the name of the job.
 *
 * @param name the new name
 */
public void setName(String name) {
    this.name = name;
}

/**
 * Gets the name of the job.
 */
public String getName() {
    return this.name;
}

/**
 * Gets the unique identifier of the job.
 */
public UUID getId() {
    return this.id;
}

/**
 * Gets the first note of the range.
 */
```

```
public int getFromNote() {
    return this.fromNote;
}

/**
 * Sets the first note of the range.
 */
public void setFromNote(int note) {
    if (note < 0 || note > 127) {
        throw new IllegalArgumentException("Note must be 0...127");
    }
    this.fromNote = note;
}

/**
 * Gets the last note of the range.
 */
public int getToNote() {
    return this.toNote;
}

/**
 * Sets the last note of the range.
 */
public void setToNote(int note) {
    if (note < 0 || note > 127) {
        throw new IllegalArgumentException("Note must be 0...127");
    }
    this.toNote = note;
}

/**
 * Gets the interval between the notes.
 */
public Interval getInterval() {
```

```
    return this.interval;
}

/**
 * Sets the interval between the notes.
 */
public void setInterval(Interval i) {
    this.interval = i;
}

/**
 * Gets the note duration.
 *
 * @return the note duration in milliseconds
 */
public int getNoteDuration() {
    return this.noteDuration;
}

/**
 * Sets the duration of each note.
 *
 * @param duration note duration (milliseconds)
 */
public void setNoteDuration(int duration) {
    if (duration <= 0) {
        throw new IllegalArgumentException("Note duration must be positive");
    }

    this.noteDuration = duration;
}

/**
 * Gets the note decay time.
 *
```

```
     * @return the note decay time in milliseconds
     */
    public int getNoteDecay() {
        return this.noteDecay;
    }

    /**
     * Sets the decay of each note.
     *
     * @param decay note decay (milliseconds)
     */
    public void setNoteDecay(int decay) {
        if (decay <= 0) {
            throw new IllegalArgumentException("Note decay time must be positive");
        }

        this.noteDecay = decay;
    }

    /**
     * Gets the gap time between notes.
     *
     * @return the note gap time in milliseconds
     */
    public int getNoteGap() {
        return this.noteGap;
    }

    /**
     * Sets the gap time between notes.
     *
     * @param gap note gap time (milliseconds)
     */
    public void setNoteGap(int gap) {
        if (gap <= 0) {
```

```
            throw new IllegalArgumentException("Note gap time must be positive");
        }

        this.noteGap = gap;
    }

    public List<Integer> getNotes() {
        List<Integer> notes = new ArrayList<>();

        int note = this.fromNote;
        while (note <= this.toNote) {
            notes.add(note);
            note += this.interval.getValue();
        }

        return notes;
    }

    public List<Integer> getVelocities() {
        List<Integer> clone = new ArrayList<Integer>(this.velocities.size());
        for (Integer i : this.velocities) {
            clone.add(i);
        }
        return clone;
    }

    /**
     * Sets a singular velocity. Clears any existing velocities
     * and replaces them with this one.
     *
     * @param v the velocity to set
     */
    public void setVelocity(int v) {
        if (v < 1 || v > 127) {
            throw new IllegalArgumentException("Velocity must be 1...127");
```

```
    }

    this.velocities.clear();
    this.velocities.add(v);
}

public void setSpecificVelocities(List<Integer> vs) {
    if (vs.size() == 0) {
        return;
    }

    this.velocities.clear();
    for (int v : vs) {
        // Discard any velocities not in range 1...127:
        if (v < 1 || v > 127) {
            continue;
        }
        // Discard any duplicates
        if (this.velocities.contains(v)) {
            continue;
        }
        this.velocities.add(v);
    }

    // Keep sorted, ascending
    Collections.sort(this.velocities);
}

public void setDistributedVelocities(int first, int last, int count) {
    if (first <= 0) {
        throw new IllegalArgumentException("First velocity must be positive");
    }
    if (first > 127) {
        throw new IllegalArgumentException("First velocity can be at most 127");
    }
```

```
    if (last <= 0) {
        throw new IllegalArgumentException("Last velocity must be positive");
    }
    if (last > 127) {
        throw new IllegalArgumentException("Last velocity can be at most 127");
    }
    if (first > last) {
        throw new IllegalArgumentException("First velocity must be smaller than last");
    }
    if (count < 1) {
        throw new IllegalArgumentException("Count must be one or more");
    }

    List<Integer> result = new ArrayList<Integer>();

    // double step = (double)(last - first) / (count -1);
    // double vel = first;
    // do {
    //     result.add((int)Math.round(vel));
    //     vel += step;
    // } while (vel <= last);
    // this method may result in a "ragged" distribution due to ceil up/floor down

    int step = (last - first) / (count - 1);
    for (int i = 0; i <= count - 2; i++) {
        result.add(first + (i * step));
    }
    result.add(last);
    // job.setDistributedVelocities(10,20,7);
    // velocities: 10 11 12 13 14 15 20
    // this method makes sure first and last always in the list

    this.velocities = result;
}
```

```
/**
 * Gets a string representation of the job.
 */
@Override
public String toString() {
    StringBuilder velocitiesString = new StringBuilder();
    for (int v : this.getVelocities()) {
        velocitiesString.append(v);
        velocitiesString.append(" ");
    }

    return String.format(
        "%s: from %d to %d by %d semitones, duration %d ms, decay %d ms, gap %d ms, velocities: %s",
        this.getName(),
        this.getFromNote(),
        this.getToNote(),
        this.interval.getValue(),
        this.getNoteDuration(),
        this.getNoteDecay(),
        this.getNoteGap(),
        velocitiesString.toString()
    );
}

//
// Private fields
//

private String name;
private UUID id;
private int fromNote;
private int toNote;
private Interval interval;
private int noteDuration;  // milliseconds
private int noteDecay;  // note decay time in ms
```

```
    private int noteGap;  // note gap time in ms
    private List<Integer> velocities;  // note velocities
}
```

5.6 Note.java

```
public class Note {

    private int number; // 0...127
    private int velocity; // 1...127
    private int startTime; // ms
    private int endTime; // ms

    public Note() { }

    public Note(int number, int velocity, int startTime, int endTime) {
        this.number = number;
        this.velocity = velocity;
        this.startTime = startTime;
        this.endTime = endTime;
    }

    // Implement getters and setters

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public int getVelocity() {
        return velocity;
    }

    public void setVelocity(int velocity) {
```

```
        this.velocity = velocity;
    }

    public int getStartTime() {
        return startTime;
    }

    public void setStartTime(int startTime) {
        this.startTime = startTime;
    }

    public int getEndTime() {
        return endTime;
    }

    public void setEndTime(int endTime) {
        this.endTime = endTime;
    }
}
```

5.7 NoteTable.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.stage.Stage;
import javafx.scene.Scene;

public class NoteTable extends Application {
    private Job job;  // Job instance
```

```
// this.fromNote = 40;
// this.toNote = 120;
// this.interval = Interval.SIX;
// this.noteDuration = 1000;
// this.noteDecay = 500;
// this.noteGap = 100;

@Override
public void start(Stage stage) {
    job = new Job("Test"); // Initialize job instance
    // Set velocities for the job
    List<Integer> vs = Arrays.asList(60, 80, 100);
    job.setSpecificVelocities(vs);
    System.out.println("Job:\n" + job);

    // Retrieve the list of note numbers from the job
    List<Integer> noteNumbers = job.getNotes();

    // Retrieve the list of velocities from the job
    List<Integer> velocities = job.getVelocities();

    // Initialize an ArrayList to store the Note objects
    List<Note> notes = new ArrayList<>();

    // Retrieve the duration, decay, and gap values from the job
    int noteDuration = job.getNoteDuration();
    int noteDecay = job.getNoteDecay();
    int noteGap = job.getNoteGap();

    // Initialize the current time to 0
    int currentTime = 0;

    // Iterate over each note number
    for (Integer noteNumber : noteNumbers) {
```

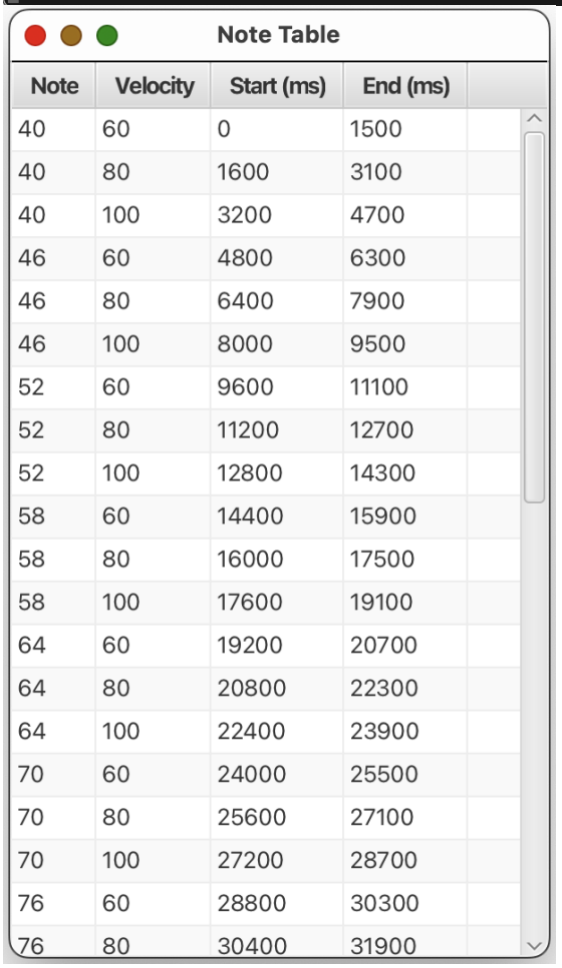
```
        // Iterate over each velocity
        for (Integer velocity : velocities) {
            // Create a new Note object
            Note note = new Note();

            // Set the note number
            note.setNumber(noteNumber);

            // Set the note velocity
            note.setVelocity(velocity);

            // Set the start time of the note to the current time
            note.setStartTime(currentTime);

            // Set the end time of the note to the start time plus note duration and decay
            note.setEndTime(currentTime + noteDuration + noteDecay);

            // Add the note to the list of notes
            notes.add(note);

            // Update the current time for the next note by adding note duration, decay and gap
            currentTime += noteDuration + noteDecay + noteGap;
        }
    }

    // Convert the ArrayList of Note objects to an ObservableList
    ObservableList<Note> noteList = FXCollections.observableArrayList(notes);


    // Create a TableView object to display the notes
    TableView<Note> table = new TableView<>();

    // This TableView should have four TableColumn objects
    TableColumn<Note, Integer> column1 = new TableColumn<>("Note");
    TableColumn<Note, Integer> column2 = new TableColumn<>("Velocity");
    TableColumn<Note, Integer> column3 = new TableColumn<>("Start (ms)");
```

```
        TableColumn<Note, Integer> column4 = new TableColumn<>("End (ms)");

        // Set the cell value factory for each TableColumn
        column1.setCellValueFactory(new PropertyValueFactory<>("number"));
        column2.setCellValueFactory(new PropertyValueFactory<>("velocity"));
        column3.setCellValueFactory(new PropertyValueFactory<>("startTime"));
        column4.setCellValueFactory(new PropertyValueFactory<>("endTime"));

        // Set the ObservableList of Note objects as the data source for the TableView
        table.setItems(noteList);

        // Add the TableColumn objects to the TableView
        table.getColumns().add(column1);
        table.getColumns().add(column2);
        table.getColumns().add(column3);
        table.getColumns().add(column4);

        //
        // Scene
        //
        Scene scene = new Scene(table, 300, 500);
        stage.setTitle("Note Table"); // Create Scene
        stage.setScene(scene);  // Set the scene
        stage.show();  // Show the stage
    }

    public static void main(String[] args) {
        launch();  // Launch the application
    }
}
```

5.8 Testing

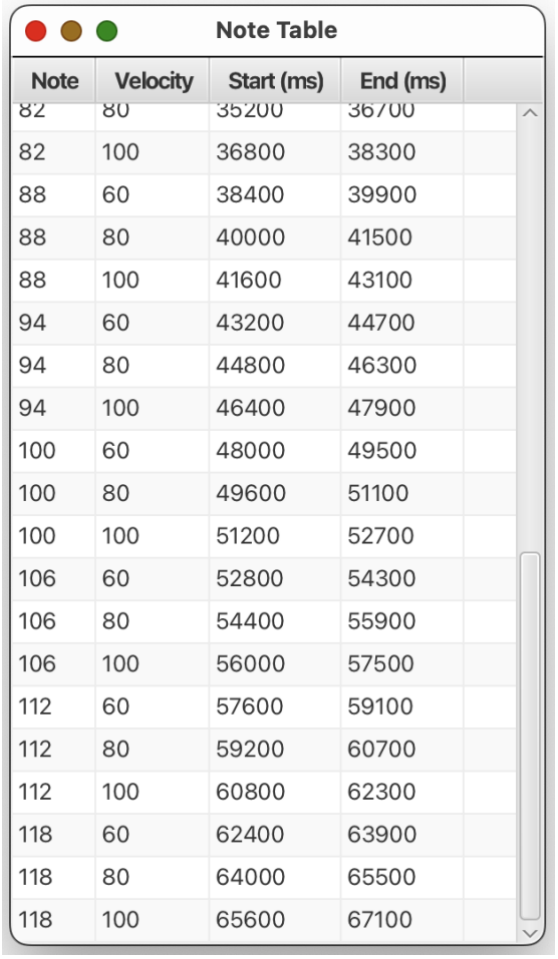
Note table for a job ranging from note 40 to 120 with an interval of six semitones, each having a duration of 1000 milliseconds, a decay time of 500 milliseconds, a gap of 100 milliseconds, and played at velocities of 60, 80, and 100:

```
jingjingyang@mac ex5 % javac --module-path $PATH_TO_FX --add-modules javafx.controls NoteTable.java
jingjingyang@mac ex5 % java --module-path $PATH_TO_FX --add-modules javafx.controls NoteTable.java
Job:
Test: from 40 to 120 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 60 80 100
```



Note Table

Note	Velocity	Start (ms)	End (ms)
40	60	0	1500
40	80	1600	3100
40	100	3200	4700
46	60	4800	6300
46	80	6400	7900
46	100	8000	9500
52	60	9600	11100
52	80	11200	12700
52	100	12800	14300
58	60	14400	15900
58	80	16000	17500
58	100	17600	19100
64	60	19200	20700
64	80	20800	22300
64	100	22400	23900
70	60	24000	25500
70	80	25600	27100
70	100	27200	28700
76	60	28800	30300
76	80	30400	31900



Note Table

Note	Velocity	Start (ms)	End (ms)
82	80	35200	36700
82	100	36800	38300
88	60	38400	39900
88	80	40000	41500
88	100	41600	43100
94	60	43200	44700
94	80	44800	46300
94	100	46400	47900
100	60	48000	49500
100	80	49600	51100
100	100	51200	52700
106	60	52800	54300
106	80	54400	55900
106	100	56000	57500
112	60	57600	59100
112	80	59200	60700
112	100	60800	62300
118	60	62400	63900
118	80	64000	65500
118	100	65600	67100

6 Final project

6.1 Overview

This application is to manage a list of musical jobs, where each job has several notes with specific parameters. The user can select a job, edit its parameters, such as velocity, duration, decay, and gap, and the application will update the list and the job's details in real-time.

6.2 Application Layout and Features

In the Main.java, I created two layouts:

setupStageTopBottom(primaryStage); // Final stage setup option 1: top, right bottom, left bottom

setupStageLeftRight(primaryStage); // Final stage setup option 2: left, top right, bottom right

I chose top, right bottom, left bottom as the final design. You can comment this out and uncomment the other in the start method to switch to the other layout.

The application window is divided into three main panes:

1. **Top Pane**: Displays a **ListView** containing a list of jobs.
2. **Bottom Left Pane**: Acts as the job editing view using **GridPane**. Center-top alignment is used to ensure that the content stays at the top and centered, regardless of changes in pane size. Changes here trigger updates not only in the job attributes but also refresh the **ListView** and **TableView** as necessary (job name change only refresh **ListView**).
3. **Bottom Right Pane**: Contains a **TableView** that lists individual notes of the selected job, displaying details such as note number, velocity, start time, and end time. Custom cell factories are implemented to align text within table columns properly.

The panes are organized using a **SplitPane** for easy resizing:

- A horizontal split allows resizing between the top list of jobs and the bottom detailed view.
- A vertical split divides the bottom section into the job editing area and the notes table.

6.3 JavaFX controls of job editing view

- **TextField** allows for job name changes.
- **Spinners** allow users to set and adjust values for note attributes (from note, to note).
- **Radio buttons** allow selection among different note intervals.
- **Sliders** allow users to adjust the duration, decay, and gap of notes. They are initialized with specific ranges and default values.
- **Checkbox** allows switching between default note times and customized ones.
- **Radio buttons** allow selection among different velocity settings. For the singular mode, a slider is used. In the specific velocities mode, a text

field is used to input the velocities. For the distributed mode, three spinners are used to input the parameters.

- **Canvas** visually represents the cumulative effects of duration, decay, and gap. A gradient color is used in the decay rectangle to visually differentiate and illustrate the transition from the note duration to the gap.
- In addition, **Labels** provide textual descriptions for various UI elements. I've set text wrapping within the reminder labels and established a minimum height to ensure they always display all the text.
- **Toggle groups** are used for radio buttons to ensure only one option can be selected at a time. **TitledPanes** are used to create collapsible sections in the interface for better organization and usability. **ScrollPane** is used to make the job editing view scrollable.
- **HBox** and **VBox** are used for horizontal and vertical layouts, respectively, while **GridPane** is utilized to neatly arange related controls.

6.4 Event Handling

- **When Selection Changes**: The **ListView** selection change triggers a full update of the job editing controls and the notes table to reflect the selected job's settings.
- **When Job Details Change**:
  - The list of jobs displayed on the top pane is dynamic and updates instantly whenever a job is selected or edited, ensuring that all changes are saved and reflected across the application without loss of data.
  - In addition, the note table also updates immediately when any changes affecting notes occur.
  - The canvas dynamically updates as the note times are adjusted, providing immediate feedback on the temporal characteristics of the notes. This is achieved through a method that redraws the segments representing duration, decay, and gap whenever any note time value changes.

6.5 Implementation process

- I started by setting up the **SplitPane** to organize the interface into multiple sections. Then I add initial UI components from weekly exercises and ensure that the job editing view and notes table are updated when a different job is selected.
- I refactored code and ensured that the job list view and notes table updated in real-time as changes were made to the current job by setting up detailed event listeners
- I added a visually appealing gradient to the decay attribute in the notes visualization.
- I introduced a set of controls for setting velocities, including options for singular, specific, and distributed velocities. And I switched to a top-bottom layout from a left-right layout.
- I added restrictions to the spinners to ensure the range of values remained logical.
- I added a label below the text input field for the job name. This label serves as a reminder about the length limit of the job name. If users hit Enter or Tab, the input is checked. If valid, the label states "Job name

has been updated." If not, it reads "Invalid Job Name. It should be 1-20 characters long."

- For Specific Velocities Input, only valid numerical inputs are accepted, while all others are ignored. A label below the input field reminds users that velocities should be comma-separated and each value must fall within the range of 1-127. The reminder text changes in a similar fashion to the job name input reminders. This guidance helps users understand the required input format and range constraints for specific velocities.
- I conducted thorough functionality tests, debugged as needed, made final code/comments adjustments for better readability and maintainability, and documented the application comprehensively.

6.6 Challenges Encountered

- **Data Consistency**: Ensuring that changes are reflected across different UI components without loss was challenging.
- **Code Refactoring**: Tools provided by IDEA JDK greatly assisted in the code refactoring process.
- **Implementing Real-Time Reflection**: As the final design did not include an 'update job' button, implementing real-time reflection required careful handling of interactions between various components.
- **Spinner Range Limits**: My solution is adjustments in the **fromNoteSpinner** and **toNoteSpinner** dynamically update the allowed ranges for each other to prevent invalid configurations. This solution also works well for three distributed velocity spinners.

6.7 Learnings and Moving forward

Throughout this project, I learned to build client applications with rich user interfaces using JavaFX, handle complex interactions between UI components, and manage the underlying logic through effective event management.

This project has been instrumental in advancing my JavaFX skills and has set a solid foundation for future projects in creating more sophisticated JavaFX applications.

6.8 Main.java

```
import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import java.util.regex.Pattern;

import java.util.stream.Collectors;



import javafx.application.Application;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;
```

```
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.*;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.paint.CycleMethod;
import javafx.scene.paint.LinearGradient;
import javafx.scene.paint.Stop;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class Main extends Application {

    private Job currentJob= new Job("test") ;  // Job instance

    private ObservableList<Job> jobs = FXCollections.observableArrayList();

    // Create ListView on left
    private ListView<Job> jobsListView;

    // Create GridPane and its components on right top
    private ScrollPane jobEditScrollPane = new ScrollPane();
    private GridPane jobEditPaneGrid = new GridPane();
    private TextField jobNameField = new TextField();
```

```
    private Label jobNameWarningLabel = new Label("Job name should be 1-20 characters long. \nPress Enter/Tab to
save changes.");
    private Spinner<Integer> fromNoteSpinner = new Spinner<>(1, currentJob.getToNote() - 1,
currentJob.getFromNote());
    private Spinner<Integer> toNoteSpinner = new Spinner<>(currentJob.getFromNote() + 1, 127,
currentJob.getToNote());
    private ToggleGroup internalGroup = new ToggleGroup();
    private HBox internalsHbox = new HBox(10);// Spacing of 10 pixels between each radio button
    // Labels for Sliders
    private Label durationLabel = new Label("Duration: 0 ms");
    private Label decayLabel = new Label("Decay: 0 ms");
    private Label gapLabel = new Label("Gap: 0 ms");
    // Sliders
    private Slider durationSlider = new Slider(100, 5000, 1000);// note duration min 100ms, max 5000ms
    private Slider decaySlider = new Slider(100, 4500, 500);// note decay min 100ms, max 4500ms
    private Slider gapSlider = new Slider(100, 500, 100);// note gap min 100ms, max 500ms
    // Sliders
    private CheckBox checkBox = new CheckBox("Use default times");
    private Label totaltimeLabel = new Label("Total Note Time: 0 ms");
    private Canvas noteTimeCanvas = new Canvas(300, 50);


    // Velocity selection setup
    private ToggleGroup velocityGroup = new ToggleGroup();
    private RadioButton singularVelocityButton = new RadioButton("Singular Velocity");
    private RadioButton specificVelocitiesButton = new RadioButton("Specific Velocities");
    private Label specificVelocitiesWarningLabel = new Label("Please enter comma-separated numbers within 1-127.
\nPress Enter/Tab to save changes.");
    private RadioButton distributedVelocitiesButton = new RadioButton("Distributed Velocities");
    // Velocity control Components
    private Slider singularVelocitySlider = new Slider(1, 127, 80);
    private Label singularVelocityLabel = new Label("80");
    private TextField specificVelocitiesField = new TextField();
    private Spinner<Integer> firstVelocitySpinner = new Spinner<>(1, 127, 40);
    private Spinner<Integer> lastVelocitySpinner = new Spinner<>(firstVelocitySpinner.getValue()+1, 127, 100);
```

```
    private Spinner<Integer> countSpinner = new Spinner<>(0, lastVelocitySpinner.getValue()-
firstVelocitySpinner.getValue()+1, 4);  // range for count

    // Create TableView on right bottom
    private ObservableList<Note> noteList = FXCollections.observableArrayList();
    private TableView<Note> noteTable;

    @Override
    public void start(Stage primaryStage) {
        //
        // Customize Job List
        //
        Job job1 = new Job("Job one");
        job1.setFromNote(80);
        job1.setToNote(100);
        job1.setVelocity(85);

        Job job2 = new Job("Job two");
        job2.setNoteDuration(3500);
        List<Integer> vs = Arrays.asList(60, 60, 100);
        job2.setSpecificVelocities(vs);

        Job job3 = new Job("Job three");
        job3.setNoteGap(300);
        job3.setDistributedVelocities(50, 90, 4);

        // Create a list of jobs
        jobs.addAll(job1,job2,job3);

        //
        // Left Section
        //
        jobsListView = getJobListView();

        //
```

```
        // Right Top Section
        //
        setupJobComponents();// // GridPane configuration
        updateUIWithJob(currentJob);// Initialize UI with default job


        //
        // Right Bottom Section
        //
        noteTable = getNoteTableView();



        setupListeners(); // Add listeners for job changes, slider movements, etc.


        setupStageTopBottom(primaryStage);// Final stage setup option 1: top, right bottom, left bottom
//        setupStageLeftRight(primaryStage);// Final stage setup option 2: left, top right, bottom right
    }


    private ListView<Job> getJobListView() {
        // Create a ListView for left section using a list of Jobs
        jobsListView = new ListView<>(jobs);


        //
        // Current job
        //


        // Select the first job by default
        jobsListView.getSelectionModel().selectFirst();
        currentJob = jobs.get(0);


        jobsListView.getSelectionModel().selectedItemProperty().addListener(
                (observable, oldValue, newValue) -> {
                    if (newValue != null) { // Check if the new value is not null
                        int selectedIndex = jobsListView.getSelectionModel().getSelectedIndex();
                        currentJob = newValue; // Use newValue directly instead of getting it from the list
                        velocityGroup.selectToggle(null); // Reset none of the toggles are selected
                        updateUIWithJob(currentJob); // Update UI for selected job
```

```
                }
            }
        );

        return jobsListView;
    }

    private void refreshJobListView() {
        // Trigger an update to ListView to reflect changes
        jobsListView.setItems(null);// Clear items to refresh

        jobsListView.setItems(jobs);// Set new items

        // Optionally, if jobs list is not recreated each time
        jobsListView.refresh();
    }

    private void setupIntervalToggleGroup() {
        updateIntervalToggleGroup(); // Call this method to create/update radio buttons

        TitledPane intervalTP = new TitledPane("Interval", internalsHbox);
        intervalTP.setCollapsible(true);
        jobEditPaneGrid.add(intervalTP, 0, 5, 2, 1);  // Adds the TitledPane to the grid.
    }

    public void updateIntervalToggleGroup() {
        internalsHbox.getChildren().clear(); // Clear existing radio buttons

        List<Job.Interval> intervals = Arrays.asList(Job.Interval.ONE, Job.Interval.THREE, Job.Interval.SIX,
Job.Interval.TWELVE);
        for (Job.Interval interval : intervals) {
            RadioButton button = new RadioButton(interval.name());
            button.setToggleGroup(internalGroup);

            // Check if interval is the current job's interval
```

```
        if (interval == currentJob.getInterval()) {
            button.setSelected(true);
        }

        // Set an action on the radio button to update current job's interval
        button.setOnAction(event -> {
            currentJob.setInterval(interval);

            refreshJobListView();// update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change
        });

        internalsHbox.getChildren().add(button); // Add the button to the HBox
    }
}

private void setupSliders() {
    // set slider properties
    durationSlider.setShowTickMarks(true);// display tick marks on the slider
    decaySlider.setShowTickMarks(true);
    gapSlider.setShowTickMarks(true);
    durationSlider.setShowTickLabels(true);// show labels for the tick marks
    decaySlider.setShowTickLabels(true);
    gapSlider.setShowTickLabels(true);
    durationSlider.setMajorTickUnit(100);// set the unit distance between major tick marks
    decaySlider.setMajorTickUnit(100);
    gapSlider.setMajorTickUnit(10);
    durationSlider.setBlockIncrement(100);// thumb move when using arrow keys
    decaySlider.setBlockIncrement(100);
    gapSlider.setBlockIncrement(10);

    VBox vbox = new VBox();
    vbox.setSpacing(10); // Sets the space to 10
    vbox.getChildren().addAll(durationLabel, durationSlider, decayLabel, decaySlider, gapLabel, gapSlider);
```

```
        TitledPane noteTimesTP = new TitledPane("Note Times", vbox);
        noteTimesTP.setCollapsible(true);
        noteTimesTP.setExpanded(false);// set this TitledPane to be collapsed by default when your application starts
        jobEditPaneGrid.add(noteTimesTP, 0, 6, 2, 1);  // Adds the TitledPane to the grid at column 0, row 5, and makes it span across 2 columns and 1 row.
    }

    private void setupCheckBoxAndCanvas() {
        //
        // CheckBox
        //
        jobEditPaneGrid.add(checkBox, 0, 7, 2, 1);  // Adds the checkBox to the grid at column 0, row 6, and makes it span across 2 columns and 1 row.

        //
        // Label to show total note time
        //
        jobEditPaneGrid.add(totaltimeLabel, 0, 8, 2, 1);  // Adds the time label to the grid at column 0, row 7, and makes it span across 2 columns and 1 row.

        //
        // Canvas
        //
        updateCanvas(); // initializes the canvas
        jobEditPaneGrid.add(noteTimeCanvas, 0, 9, 2, 1);  // Adds the canvas to the grid at column 0, row 8, and makes it span across 2 columns and 1 row.
    }

    private void setupVelocityToggleGroup() {
        GridPane velocityGrid = new GridPane();
        velocityGrid.setAlignment(Pos.TOP_CENTER);
        velocityGrid.setHgap(10);
        velocityGrid.setVgap(15);
        velocityGrid.setPadding(new Insets(25));
```

```
        singularVelocityButton.setToggleGroup(velocityGroup);
        specificVelocitiesButton.setToggleGroup(velocityGroup);
        distributedVelocitiesButton.setToggleGroup(velocityGroup);

        // Singular Velocity Components
        velocityGrid.add(singularVelocityButton, 0, 0);
        velocityGrid.add(singularVelocitySlider, 1, 0);
        velocityGrid.add(singularVelocityLabel, 1, 1);

        // Specific Velocities Components
        velocityGrid.add(specificVelocitiesButton, 0, 2);
        velocityGrid.add(specificVelocitiesField, 1, 2);
        velocityGrid.add(specificVelocitiesWarningLabel, 1, 3);
        specificVelocitiesWarningLabel.setWrapText(true);  // Enable text wrapping within the label
        specificVelocitiesWarningLabel.setMinHeight(Control.USE_PREF_SIZE);  // Set minimum height to use preferred
size

        Label firstVelocityLabel = new Label("First Velocity:");
        Label lastVelocityLabel = new Label("Last Velocity:");
        Label countLabel = new Label("Count:");

        // Distributed Velocities Components
        velocityGrid.add(distributedVelocitiesButton, 0, 4,2,1);
        velocityGrid.add(firstVelocityLabel, 0, 5);
        velocityGrid.add(firstVelocitySpinner, 1, 5);
        velocityGrid.add(lastVelocityLabel, 0, 6);
        velocityGrid.add(lastVelocitySpinner, 1, 6);
        velocityGrid.add(countLabel, 0, 7);
        velocityGrid.add(countSpinner, 1, 7);


        singularVelocityLabel.setAlignment(Pos.CENTER);
        firstVelocityLabel.setAlignment(Pos.CENTER_RIGHT);
        lastVelocityLabel.setAlignment(Pos.CENTER_RIGHT);
        countLabel.setAlignment(Pos.CENTER_RIGHT);
```

```
        // set the minimum or preferred width to see the effect
        singularVelocityLabel.setMinWidth(200);
        firstVelocityLabel.setMinWidth(120);
        lastVelocityLabel.setMinWidth(120);
        countLabel.setMinWidth(120);

        TitledPane velocityTP = new TitledPane("Note Velocities", velocityGrid);
        velocityTP.setCollapsible(true);
        velocityTP.setExpanded(false);
        jobEditPaneGrid.add(velocityTP, 0, 10, 2, 1);
    }

    private void setupJobComponents() {
        jobEditPaneGrid.setAlignment(Pos.TOP_CENTER);  // Center align the GridPane
        jobEditPaneGrid.setHgap(10);  // Set horizontal gap between grid cells
        jobEditPaneGrid.setVgap(15);  // Set vertical gap between grid cells
        jobEditPaneGrid.setPadding(new Insets(25));  // Set the padding for the GridPane to 25 units on all sides
        jobEditPaneGrid.setMinSize(0, 0);  // Set minimum width and height to 0

        //
        // Text for heading
        //
        Text heading = new Text("Please customize the job details");
        heading.setFont(new Font(20)); // Set font size to 20
        jobEditPaneGrid.add(heading, 0, 0, 2, 1);  // Adds the heading to the grid at column 0, row 0, and makes it span
across 2 columns and 1 row.
        GridPane.setHalignment(heading, HPos.CENTER);

        //
        // Labels
        //
        Label jobNameLabel = new Label("Job Name:");  // Label for the job name field
        Label fromNoteLabel = new Label("From Note:");  // Label for the from note spinner
        Label toNoteLabel = new Label("To Note:");  // Label for the to note spinner
        jobEditPaneGrid.add(jobNameLabel, 0, 1);
```

```
    jobEditPaneGrid.add(jobNameWarningLabel, 1, 2, 2, 1);
    jobNameWarningLabel.setWrapText(true);  // Enable text wrapping within the label
    jobNameWarningLabel.setMinHeight(Control.USE_PREF_SIZE);  // Set minimum height to use preferred size

    jobEditPaneGrid.add(fromNoteLabel, 0, 3);
    jobEditPaneGrid.add(toNoteLabel, 0, 4);

    //
    // TextField and Spinners
    //
    jobEditPaneGrid.add(jobNameField, 1, 1);
    jobEditPaneGrid.add(fromNoteSpinner, 1, 3);
    jobEditPaneGrid.add(toNoteSpinner, 1, 4);

    setupIntervalToggleGroup();// Setup RadioButtons for intervals
    setupSliders();
    setupCheckBoxAndCanvas();
    setupVelocityToggleGroup();

    jobEditScrollPane.setContent(jobEditPaneGrid);
    jobEditScrollPane.setFitToWidth(true);  // Ensures the grid expands to fill the scroll pane width
    jobEditScrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);  // Vertical scroll bar
}

private void updateCanvas() {
    GraphicsContext gc = noteTimeCanvas.getGraphicsContext2D();

    // Calculate the total
    int total = currentJob.getNoteDuration() + currentJob.getNoteDecay() + currentJob.getNoteGap();
    totaltimeLabel.setText("Total Note Time: " + total + " ms");

    // Calculate the percentage of each attribute
    double durationPercentage = (double) currentJob.getNoteDuration() / total;
    double decayPercentage = (double) currentJob.getNoteDecay() / total;
    double gapPercentage = (double) currentJob.getNoteGap() / total;
```

```
// Calculate the starting point for each rectangle
double durationWidth = durationPercentage * noteTimeCanvas.getWidth();
double decayWidth = decayPercentage * noteTimeCanvas.getWidth();
double decayStartX = durationWidth; // The decay rectangle starts where the duration rectangle ends

// Draw the duration rectangle with a solid color
gc.setFill(Color.TEAL);
gc.fillRect(0, 0, durationWidth, 50);

// Create a LinearGradient for the decay rectangle
Stop[] decayGradientStops = new Stop[] {
        new Stop(0, Color.TEAL), // Start color of the gradient
        new Stop(1, Color.BEIGE)  // End color of the gradient
};

// Create a linear gradient
LinearGradient decayGradient = new LinearGradient(
        0, 0, 1, 0,
        true,
        CycleMethod.NO_CYCLE,
        decayGradientStops
);

// Draw the decay rectangle with the gradient
gc.setFill(decayGradient);
gc.fillRect(decayStartX, 0, decayWidth, 50);

// Draw the gap rectangle with a solid color
double gapStartX = decayStartX + decayWidth; // It starts where the decay rectangle ends
double gapWidth = gapPercentage * noteTimeCanvas.getWidth();
gc.setFill(Color.BEIGE);
gc.fillRect(gapStartX, 0, gapWidth, 50);

// Set the stroke color
```

```
    gc.setStroke(Color.BLACK); // This will set the color of the borders

    // Set the line width for the borders
    gc.setLineWidth(1); // This will set the border width to 1 pixel

    // Draw borders around the rectangles
    gc.strokeRect(0, 0, durationWidth, 50); // Border for duration rectangle
    gc.strokeRect(decayStartX, 0, decayWidth, 50); // Border for decay rectangle
    gc.strokeRect(gapStartX, 0, gapWidth, 50); // Border for gap rectangle

    // Set the fill color for the text
    gc.setFill(Color.BLACK);

    // Add text to the rectangles
    gc.fillText("Duration", 5, 30);
    gc.fillText("Decay", durationPercentage * noteTimeCanvas.getWidth() + 5, 30);
    gc.fillText("Gap", (durationPercentage + decayPercentage) * noteTimeCanvas.getWidth() + 5, 30);
}

private void updateUIWithJob(Job job) {
    // Update UI components based on updated job
    jobNameField.setText(job.getName());
    fromNoteSpinner.getValueFactory().setValue(job.getFromNote());
    toNoteSpinner.getValueFactory().setValue(job.getToNote());

    updateIntervalToggleGroup(); // update the radio buttons based on the current job

    durationLabel.setText("Duration: " + currentJob.getNoteDuration() + "ms");
    decayLabel.setText("Decay: " + currentJob.getNoteDecay() + "ms");
    gapLabel.setText("Gap: " + currentJob.getNoteGap() + "ms");

    durationSlider.setValue(job.getNoteDuration());
    decaySlider.setValue(job.getNoteDecay());
    gapSlider.setValue(job.getNoteGap());
```

```
    updateCanvas();
}

private List<Note> getNotes() {
    // Retrieve the list of note numbers from the job
    List<Integer> noteNumbers = currentJob.getNotes();

    // Retrieve the list of velocities from the job
    List<Integer> velocities = currentJob.getVelocities();

    // Initialize an ArrayList to store the Note objects
    List<Note> notes = new ArrayList<>();

    // Retrieve the duration, decay, and gap values from the job
    int noteDuration = currentJob.getNoteDuration();
    int noteDecay = currentJob.getNoteDecay();
    int noteGap = currentJob.getNoteGap();

    // Initialize the current time to 0
    int currentTime = 0;

    // Iterate over each note number
    for (Integer noteNumber : noteNumbers) {

        // Iterate over each velocity
        for (Integer velocity : velocities) {
            // Create a new Note object
            Note note = new Note();

            // Set the note number
            note.setNumber(noteNumber);

            // Set the note velocity
            note.setVelocity(velocity);
```

```
            // Set the start time of the note to the current time
            note.setStartTime(currentTime);

            // Set the end time of the note to the start time plus note duration and decay
            note.setEndTime(currentTime + noteDuration + noteDecay);

            // Add the note to the list of notes
            notes.add(note);

            // Update the current time for the next note by adding note duration, decay and gap
            currentTime += noteDuration + noteDecay + noteGap;
        }
    }
    return notes;
}

// Method to set right alignment for table column cells
private void setRightAlignedTableColumn(TableColumn<Note, Integer> column) {
    column.setCellFactory(tc -> new TableCell<Note, Integer>() {
        @Override
        protected void updateItem(Integer item, boolean empty) {
            super.updateItem(item, empty);
            if (empty || item == null) {
                setText(null);
            } else {
                setText(item.toString());
                setAlignment(Pos.CENTER_RIGHT); // Right align the text
            }
        }
    });
}

private TableView<Note> getNoteTableView() {
    // Convert the ArrayList of Note objects to an ObservableList
    noteList = FXCollections.observableArrayList(getNotes());
```

```
        //
        // table
        //
        // Create a TableView object to display the notes
        noteTable = new TableView<>(noteList);

        // This TableView should have four TableColumn objects
        TableColumn<Note, Integer> column1 = new TableColumn<>("Note");
        TableColumn<Note, Integer> column2 = new TableColumn<>("Velocity");
        TableColumn<Note, Integer> column3 = new TableColumn<>("Start (ms)");
        TableColumn<Note, Integer> column4 = new TableColumn<>("End (ms)");

        // Set the cell value factory for each TableColumn
        column1.setCellValueFactory(new PropertyValueFactory<>("number"));
        column2.setCellValueFactory(new PropertyValueFactory<>("velocity"));
        column3.setCellValueFactory(new PropertyValueFactory<>("startTime"));
        column4.setCellValueFactory(new PropertyValueFactory<>("endTime"));

        // Apply right alignment to the columns
        setRightAlignedTableColumn(column1);
        setRightAlignedTableColumn(column2);
        setRightAlignedTableColumn(column3);
        setRightAlignedTableColumn(column4);

        // Set the ObservableList of Note objects as the data source for the TableView
        noteTable.setItems(noteList);

        // Add the TableColumn objects to the TableView
        noteTable.getColumns().addAll(column1, column2, column3, column4);
        return noteTable;
    }

    private void refreshNotesTableView() {
        List<Note> notes = getNotes(); // Generate the list uses currentJob
```

```
    noteList.setAll(notes); // Replace the items in the TableView with the new list
    noteTable.refresh(); // Refresh the TableView to display the new items
}

private void setupListeners() {

    // Add an onAction event from the text field (which triggers when Enter is pressed)
    jobNameField.setOnAction(event -> {
        String newName = jobNameField.getText().trim();
        if (newName.length() > 0 && newName.length() <= 20) {
            currentJob.setName(newName);
            refreshJobListView();
            jobNameWarningLabel.setText("Job name has been updated.");
        } else {
            jobNameWarningLabel.setText("Invalid Job Name. It should be 1-20 characters long.");
        }
    });

    // Add a listener to the 'jobNameField' focused property (which triggers when Tab key is pressed)
    jobNameField.focusedProperty().addListener((observable, oldValue, newValue) -> {
        if (!newValue) { // focus lost
            String newName = jobNameField.getText().trim();
            if (newName.length() > 0 && newName.length() <= 20) {
                currentJob.setName(newName);
                refreshJobListView();
                jobNameWarningLabel.setText("Job name has been updated.");
            } else {
                jobNameWarningLabel.setText("Invalid Job Name. It should be 1-20 characters long.");
            }
        }
    });

    // Add a listener to the 'fromNoteSpinner' value property
    fromNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
        // update the minimum value of toNoteSpinner
```

```
            int newMin = newValue + 1;
            toNoteSpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(newMin, 127,
Math.max(toNoteSpinner.getValue(), newMin)));

            // update job
            currentJob.setFromNote(newValue.intValue());
            refreshJobListView();// update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change

        });

        // Add a listener to the 'toNoteSpinner' value property
        toNoteSpinner.valueProperty().addListener((observable, oldValue, newValue) -> {
            // update the maximum value of fromNoteSpinner
            int newMax = Math.max(0, newValue - 1);
            fromNoteSpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(0, newMax,
Math.min(fromNoteSpinner.getValue(), newMax)));

            // update job
            currentJob.setToNote(newValue.intValue());
            refreshJobListView();// update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change

        });

        // Add listeners to the sliders value property
        durationSlider.valueProperty().addListener((observable, oldValue, newValue) -> {
            durationLabel.setText("Duration: " + newValue.intValue() + " ms");
            // update job
            currentJob.setNoteDuration(newValue.intValue());
            refreshJobListView();// update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change
            updateCanvas(); // update the canvas
        });
        decaySlider.valueProperty().addListener((observable, oldValue, newValue) -> {
```

```
        decayLabel.setText("Decay: " + newValue.intValue() + " ms");
        // update job
        currentJob.setNoteDecay(newValue.intValue());
        refreshJobListView();// update list of jobs whenever job details change
        refreshNotesTableView();// update table of notes whenever job details change
        updateCanvas(); // update the canvas
    });
    gapSlider.valueProperty().addListener((observable, oldValue, newValue) -> {
        gapLabel.setText("Gap: " + newValue.intValue() + " ms");
        // update job
        currentJob.setNoteGap(newValue.intValue());
        refreshJobListView();// update list of jobs whenever job details change
        refreshNotesTableView();// update table of notes whenever job details change
        updateCanvas(); // update the canvas
    });

    // Add a listener to the checkBox value property
    checkBox.selectedProperty().addListener((observable, oldValue, newValue) -> {
        durationSlider.setDisable(newValue);
        decaySlider.setDisable(newValue);
        gapSlider.setDisable(newValue);

        if (newValue) {
            currentJob.setNoteDuration(1000);// default duration 1000ms
            currentJob.setNoteDecay(500);// default decay 500ms
            currentJob.setNoteGap(100);// default gap 100ms
        } else {
            // Use the latest values if newValue is false
            currentJob.setNoteDuration((int) durationSlider.getValue());
            currentJob.setNoteDecay((int) decaySlider.getValue());
            currentJob.setNoteGap((int) gapSlider.getValue());
        }

        refreshJobListView();// update list of jobs whenever job details change
        refreshNotesTableView();// update table of notes whenever job details change
```

```
    updateCanvas(); // update the canvas
});

// Add listeners to toggle group to update UI based on selection
velocityGroup.selectedToggleProperty().addListener((observable, oldVal, newVal) -> {

    if (newVal == singularVelocityButton) {
        // If the singular velocity button is selected, apply the singular velocity settings
        int currentVelocity = (int) singularVelocitySlider.getValue();
        currentJob.setVelocity(currentVelocity);
        singularVelocityLabel.setText(String.valueOf(currentVelocity));
    } else if (newVal == specificVelocitiesButton) {
        // If the specific velocities button is selected, parse and apply specific velocities
        List<Integer> velocities = parseDistributedVelocities(specificVelocitiesField.getText());
        if (velocities != null) {
            currentJob.setSpecificVelocities(velocities);
        }
    } else if (newVal == distributedVelocitiesButton) {
        // If the distributed velocities button is selected, calculate and apply distributed velocities
        try {
            currentJob.setDistributedVelocities(
                    firstVelocitySpinner.getValue(),
                    lastVelocitySpinner.getValue(),
                    countSpinner.getValue()
            );
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }

    // Always refresh UI components to reflect the current job and note settings
    refreshJobListView(); // update list of jobs whenever job details change
    refreshNotesTableView(); // update table of notes whenever job details change
});
```

```
        // Listener for singular velocity
        singularVelocitySlider.valueProperty().addListener((obs, oldVal, newVal) -> {
            //            singularVelocityButton.isSelected()==true
            if (singularVelocityButton.isSelected()) {
                currentJob.setVelocity(newVal.intValue());
                singularVelocityLabel.setText(newVal.intValue() + "");
                refreshJobListView();  // update list of jobs whenever job details change
                refreshNotesTableView();// update table of notes whenever job details change
            }
        });

        // Add an onAction event from the specific velocities field (which triggers when Enter is pressed)
        specificVelocitiesField.setOnAction(event -> {
            if (specificVelocitiesButton.isSelected()) {
                String text = specificVelocitiesField.getText().trim();
                List<Integer> velocities = parseDistributedVelocities(text);
                if (velocities != null && !velocities.isEmpty()) {
                    currentJob.setSpecificVelocities(velocities);
                    refreshJobListView();  // update list of jobs whenever job details change
                    refreshNotesTableView(); // update table of notes whenever job details change
                    specificVelocitiesWarningLabel.setText("Velocities have been updated.");
                } else {
                    specificVelocitiesWarningLabel.setText("Invalid input. Please enter comma-separated numbers within 1-
127.");
                }
            }
        });

        // Listener for focus lost event on specific velocities field (which triggers when Tab key is pressed)
        specificVelocitiesField.focusedProperty().addListener((observable, oldValue, newValue) -> {
            if (!newValue && specificVelocitiesButton.isSelected()) { // Focus is lost and specific velocities button is selected
                String text = specificVelocitiesField.getText();
                List<Integer> velocities = parseDistributedVelocities(text);
                if (velocities != null && !velocities.isEmpty()) {
                    currentJob.setSpecificVelocities(velocities);
```

```
                refreshJobListView();  // update list of jobs whenever job details change
                refreshNotesTableView(); // update table of notes whenever job details change
                specificVelocitiesWarningLabel.setText("Velocities have been updated.");
            } else {
                specificVelocitiesWarningLabel.setText("Invalid input. Please enter comma-separated numbers within 1-
127.");
            }
        }
    });

    // Listener for distributed velocities
    firstVelocitySpinner.valueProperty().addListener((obs, oldVal, newVal) -> {
        // update the minimum value of lastVelocitySpinner
        int newMin = newVal + 1;
        lastVelocitySpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(newMin, 127,
Math.max(lastVelocitySpinner.getValue(), newMin)));

        // update the maximum value of countSpinner
        int newMax = Math.max(0, lastVelocitySpinner.getValue() - newVal + 1);
        countSpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(2, newMax,
Math.min(countSpinner.getValue(), newMax)));

        //          distributedVelocitiesButton.isSelected()==true
        if (distributedVelocitiesButton.isSelected()) {
            currentJob.setDistributedVelocities(
                    newVal,
                    lastVelocitySpinner.getValue(),
                    countSpinner.getValue()
            );
            refreshJobListView();  // update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change
        }
    });

    lastVelocitySpinner.valueProperty().addListener((obs, oldVal, newVal) -> {
```

```
        // update the maximum value of lastVelocitySpinner
        int newMax = Math.max(0, newVal - 1);
        firstVelocitySpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(1, newMax,
Math.min(firstVelocitySpinner.getValue(), newMax)));

        // update the maximum value of countSpinner
        int newCountMax = Math.max(0, newVal - firstVelocitySpinner.getValue() + 1);
        countSpinner.setValueFactory(new SpinnerValueFactory.IntegerSpinnerValueFactory(2, newCountMax,
Math.min(countSpinner.getValue(), newCountMax)));

        //          distributedVelocitiesButton.isSelected()==true
        if (distributedVelocitiesButton.isSelected()) {
            currentJob.setDistributedVelocities(
                    firstVelocitySpinner.getValue(),
                    newVal,
                    countSpinner.getValue()
            );
            refreshJobListView();  // update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change
        }
    });

    countSpinner.valueProperty().addListener((obs, oldVal, newVal) -> {
        if (distributedVelocitiesButton.isSelected()) {
            currentJob.setDistributedVelocities(
                    firstVelocitySpinner.getValue(),
                    lastVelocitySpinner.getValue(),
                    newVal
            );
            refreshJobListView();  // update list of jobs whenever job details change
            refreshNotesTableView();// update table of notes whenever job details change
        }
    });
}
```

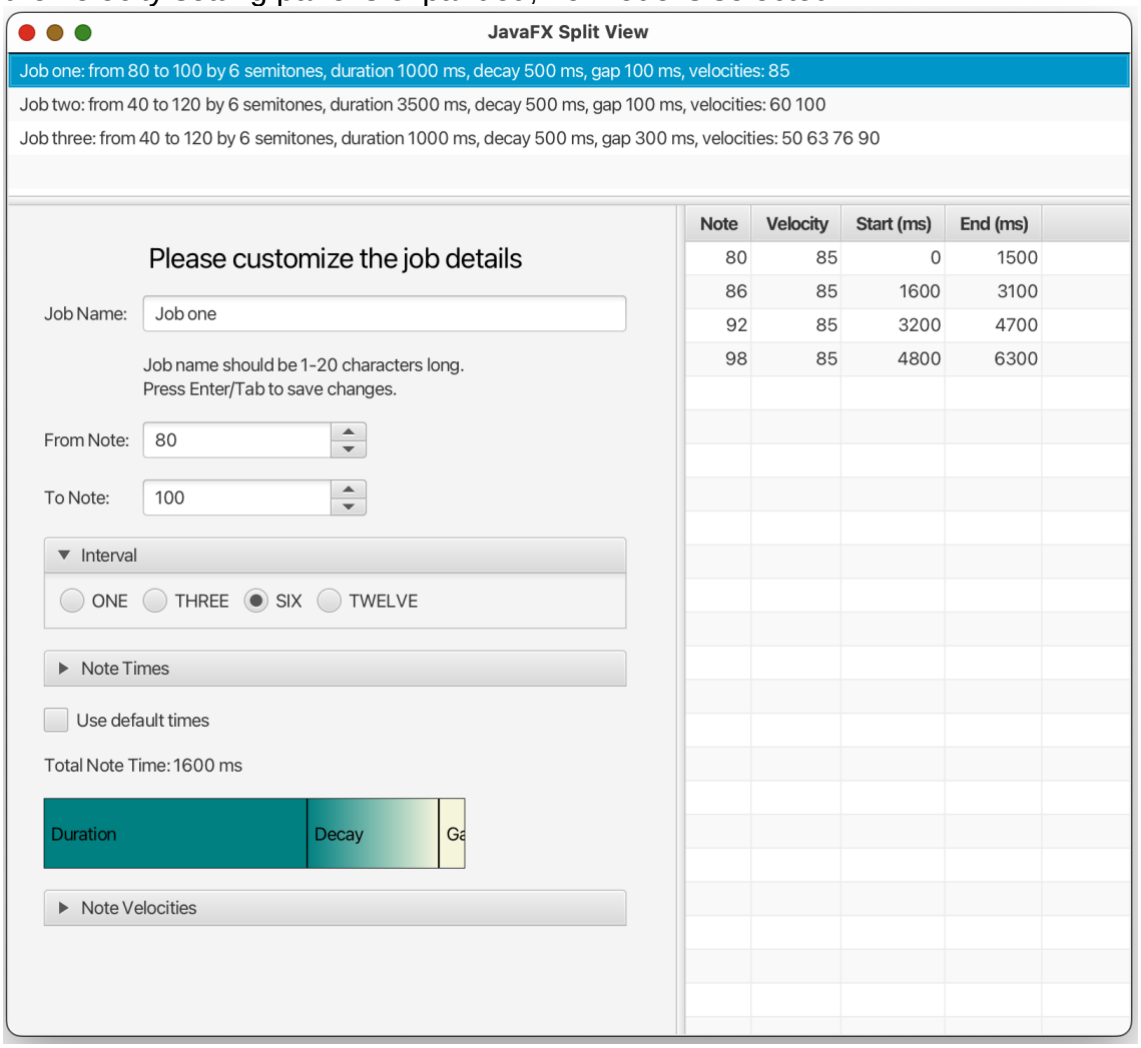
```
// Helper method to parse velocities from the text field
private List<Integer> parseDistributedVelocities(String input) {
    if (input == null || input.trim().isEmpty()) {
        return Collections.emptyList();
    }
    return Arrays.stream(input.split(","))
            .map(String::trim)
            .filter(Pattern.compile("\\d+").asPredicate())
            .map(Integer::parseInt)
            .collect(Collectors.toList());
}

private void setupStageLeftRight(Stage primaryStage) {
    //
    // Right SplitPane
    //
    // Create a SplitPane for right section
    SplitPane rightSection = new SplitPane();
    rightSection.setOrientation(Orientation.VERTICAL);
    rightSection.getItems().addAll(jobEditScrollPane, noteTable);

    //
    // Whole SplitPane
    //
    // Create a SplitPane for the whole section
    SplitPane splitPane = new SplitPane();
    // Add left section and right section to the SplitPane
    splitPane.getItems().addAll(jobsListView, rightSection);

    //
    // Scene
    //
    // Create a Scene
    Scene scene = new Scene(splitPane, 800, 500);
    // Set scene and stage
```

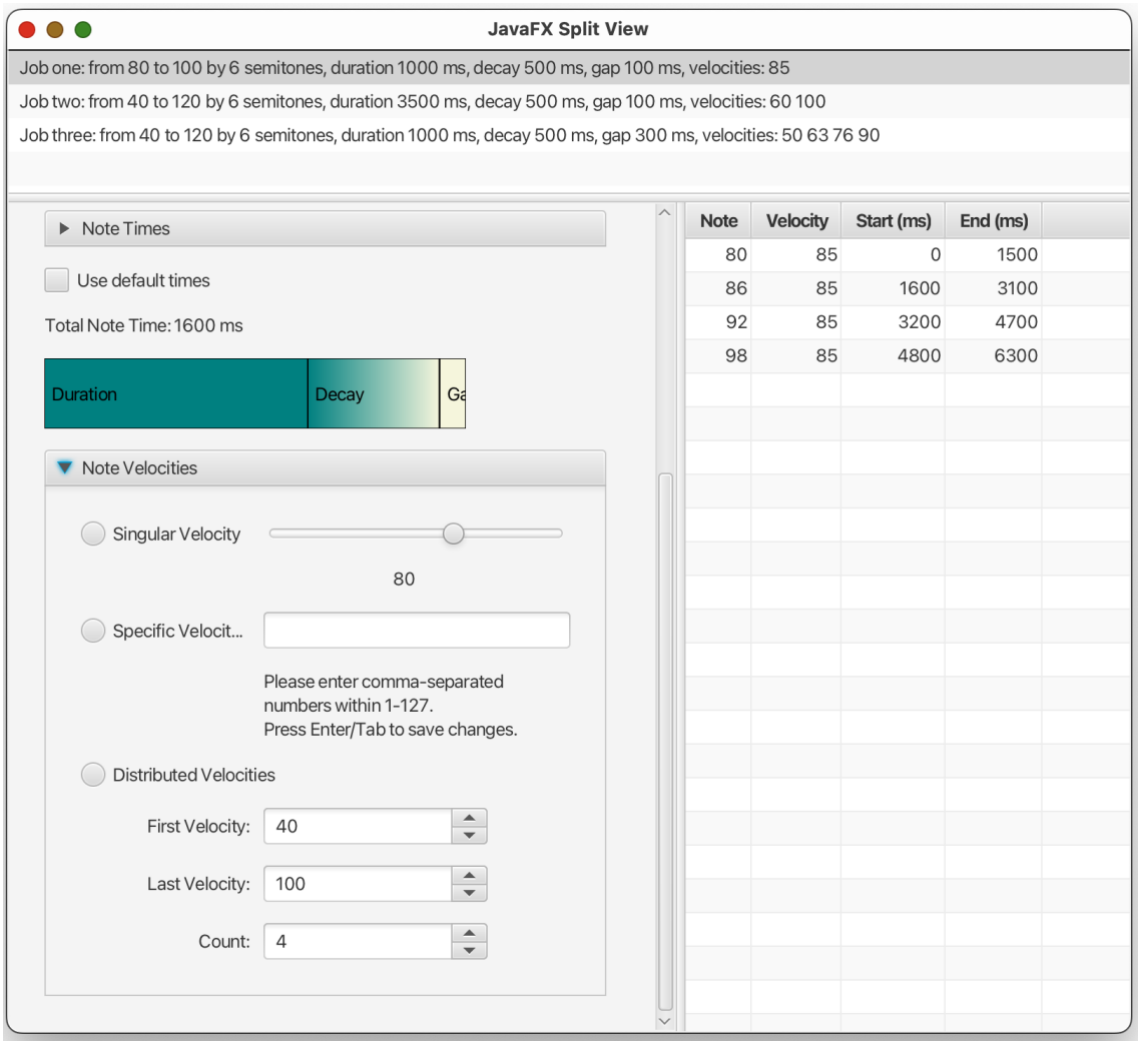
```
    primaryStage.setScene(scene);
    primaryStage.setTitle("JavaFX Split View");
    primaryStage.show();
}

private void setupStageTopBottom(Stage primaryStage) {

    //
    // Bottom SplitPane
    //
    // Create a SplitPane for the bottom section
    SplitPane bottomSection = new SplitPane();
    // Add left section and right section to the SplitPane
    bottomSection.getItems().addAll(jobEditScrollPane, noteTable);
    bottomSection.setDividerPositions(0.6);// adjust the size of each section to proper
    //
    // Whole SplitPane
    //
    // Create a SplitPane for whole section
    SplitPane topBottomSection = new SplitPane();
    topBottomSection.setOrientation(Orientation.VERTICAL);
    topBottomSection.getItems().addAll(jobsListView, bottomSection);
    topBottomSection.setDividerPositions(0.15);// adjust the size of each section to proper

    //
    // Scene
    //
    // Create a Scene
    Scene scene = new Scene(topBottomSection, 800, 700);
    // Set scene and stage
    primaryStage.setScene(scene);
    primaryStage.setTitle("JavaFX Split View");
    primaryStage.show();
}
```

```
    public static void main(String[] args) {
        launch(args);
    }
}
```

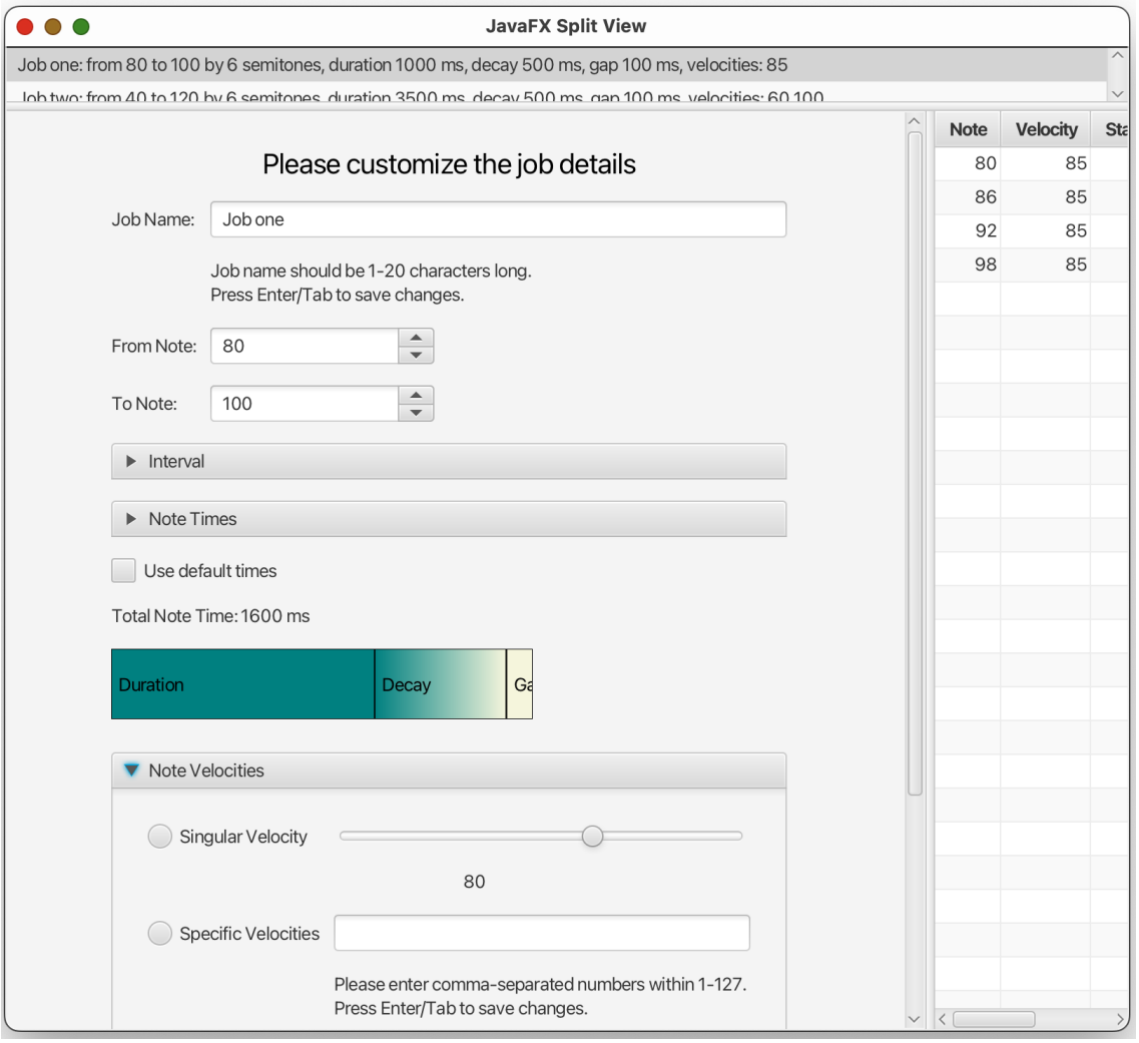
6.9 Testing

1. The initial display features the first job selected as the current job, with the note times setting pane and velocity setting pane collapsed. When the velocity setting pane is expanded, no mode is selected.

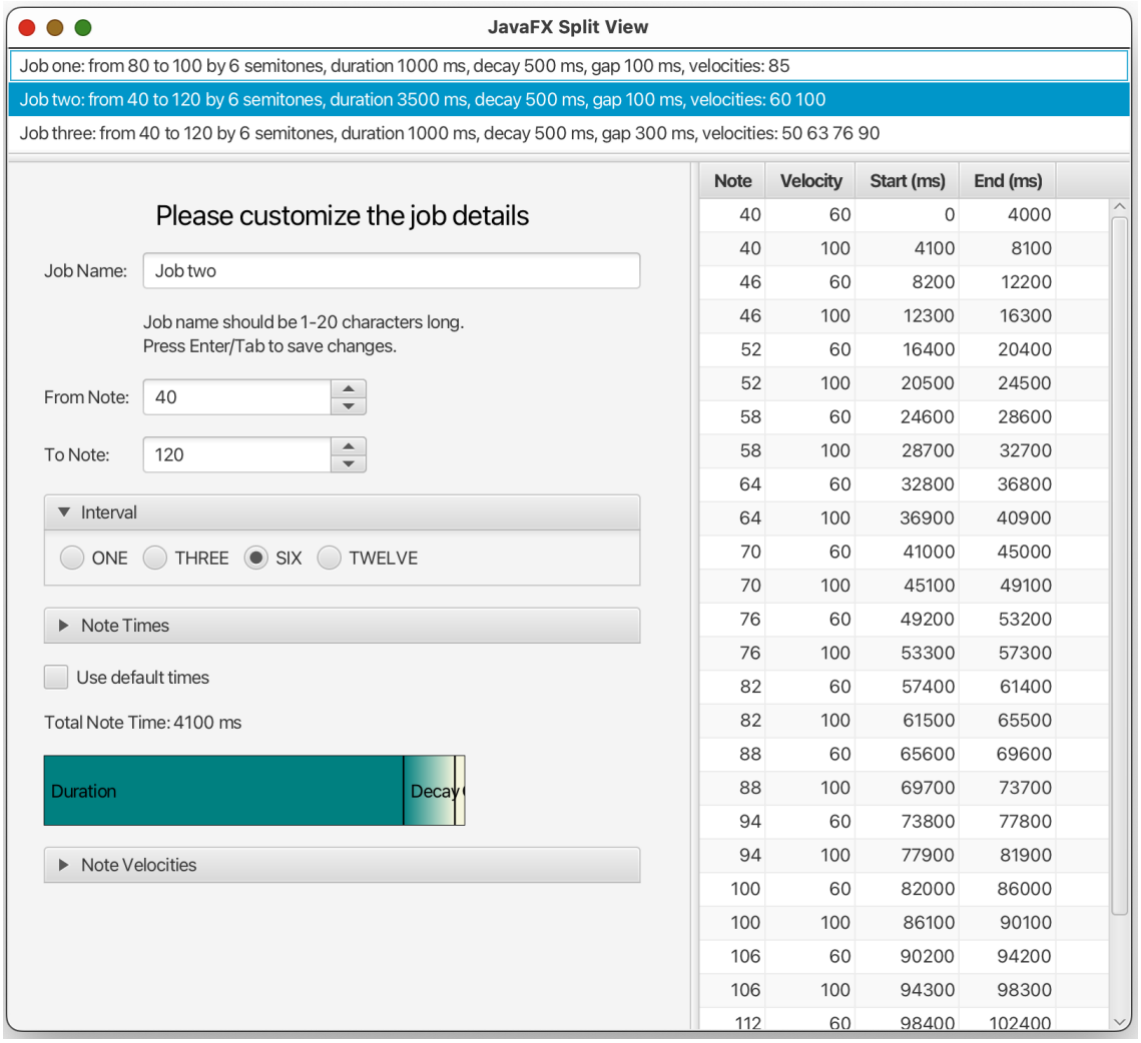


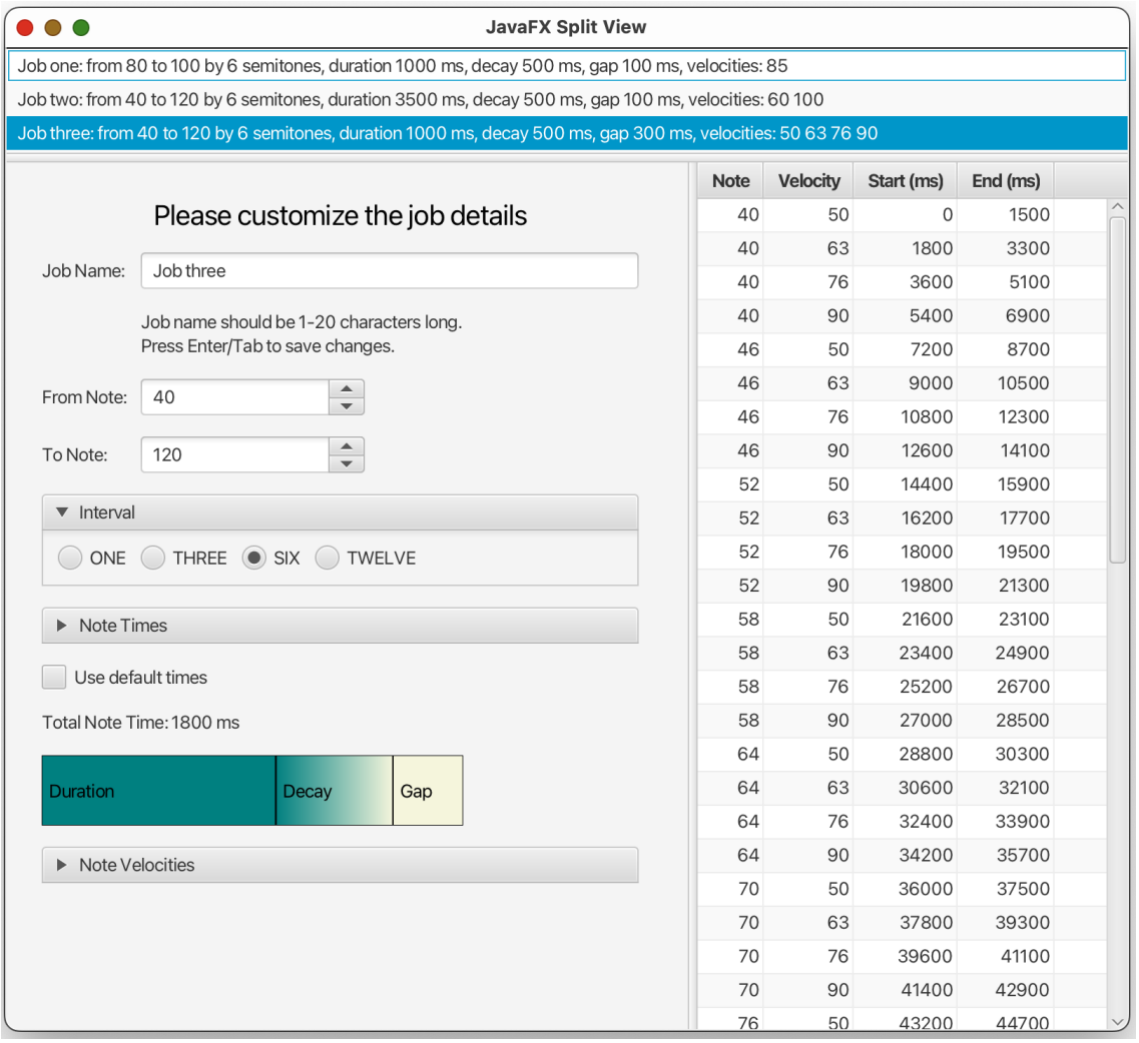

JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three: from 40 to 120 by 6 semitones, duration 1000 ms, decay 500 ms, gap 300 ms, velocities: 50 63 76 90
Note Times
Use default times
Total Note Time: 1600 ms
Duration
Decay
Ga
Note Velocities
Singular Velocity
80
Specific Velocit...
Please enter comma-separated numbers within 1-127.
Press Enter/Tab to save changes.
Distributed Velocities
First Velocity: 40
Last Velocity: 100
Count: 4
Note
Velocity
Start (ms)
End (ms)
80 85 0 1500
86 85 1600 3100
92 85 3200 4700
98 85 4800 6300

2. The three sections can be resized freely, while the job editing view remains at the top and centered, and it is scrollable vertically.

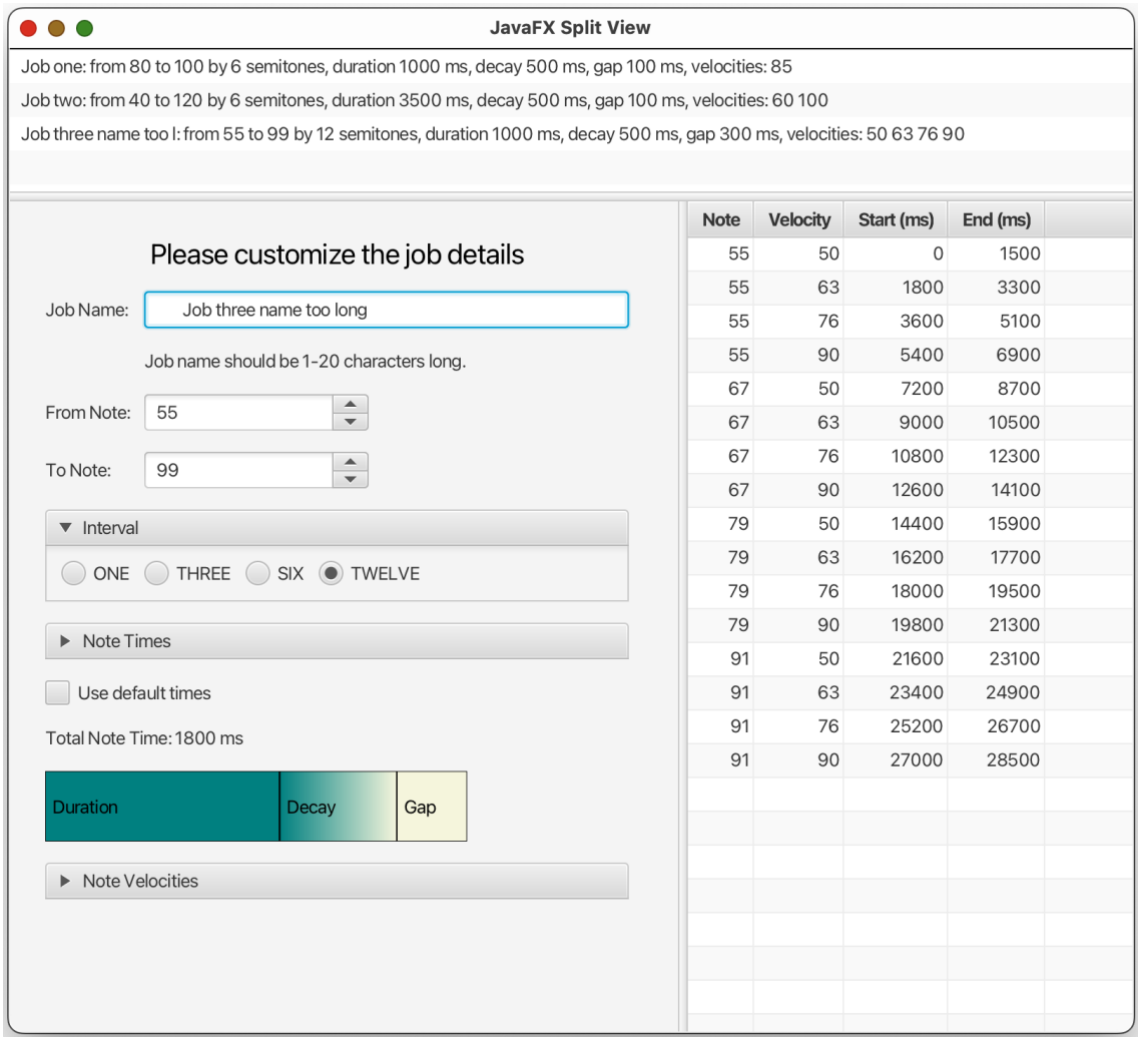


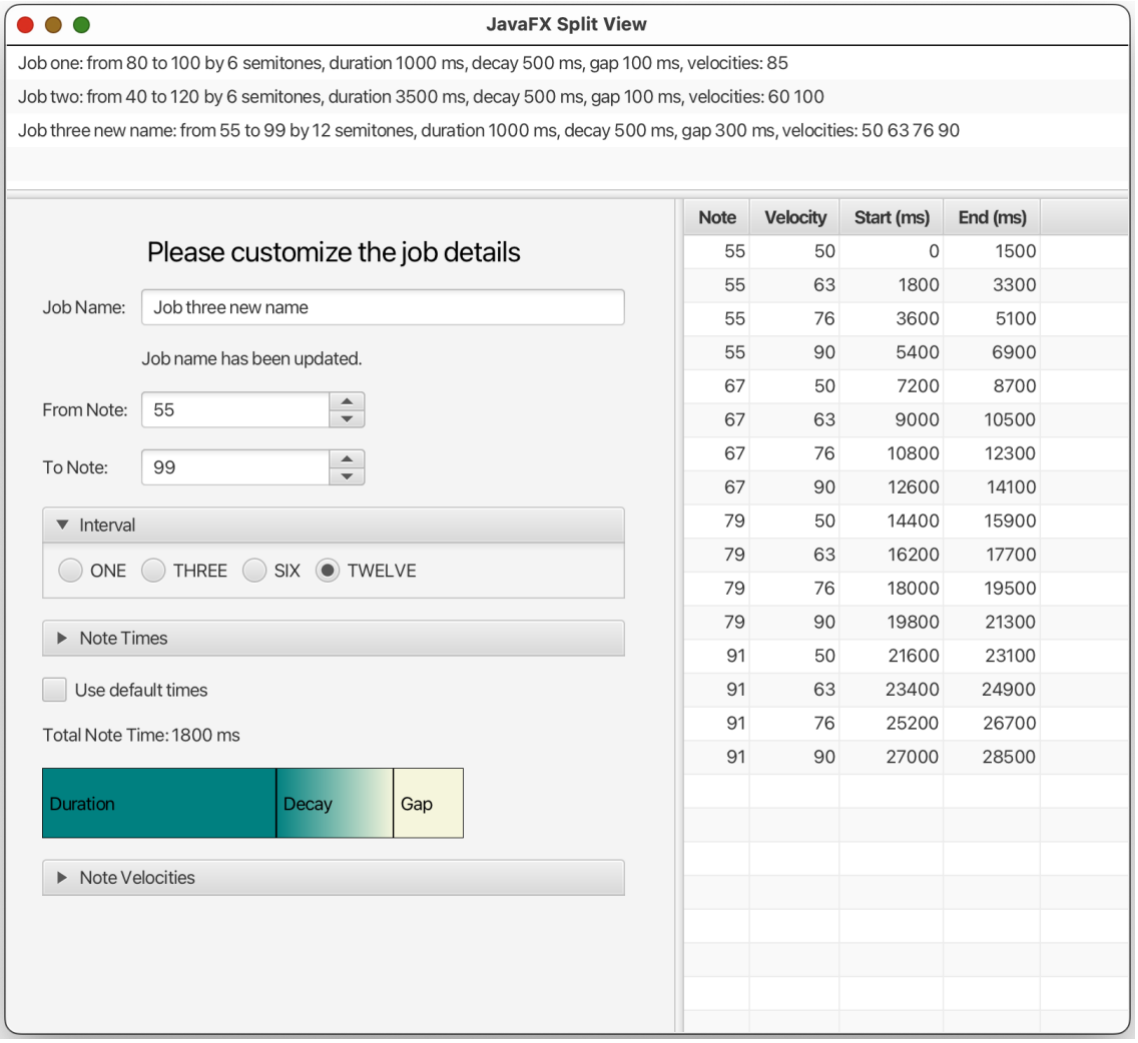
3. Upon selection of a different job, both the job editing view and the note table list view will adjust accordingly.


JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three: from 40 to 120 by 6 semitones, duration 1000 ms, decay 500 ms, gap 300 ms, velocities: 50 63 76 90
Please customize the job details
Job Name:
Job two
Job name should be 1-20 characters long.
Press Enter/Tab to save changes.
From Note:
40
To Note:
120
Interval
ONE
THREE
SIX
TWELVE
Note Times
Use default times
Total Note Time: 4100 ms
Duration
Decay
Note Velocities
Note
Velocity
Start (ms)
End (ms)
40 60 0 4000
40 100 4100 8100
46 60 8200 12200
46 100 12300 16300
52 60 16400 20400
52 100 20500 24500
58 60 24600 28600
58 100 28700 32700
64 60 32800 36800
64 100 36900 40900
70 60 41000 45000
70 100 45100 49100
76 60 49200 53200
76 100 53300 57300
82 60 57400 61400
82 100 61500 65500
88 60 65600 69600
88 100 69700 73700
94 60 73800 77800
94 100 77900 81900
100 60 82000 86000
100 100 86100 90100
106 60 90200 94200
106 100 94300 98300
112 60 98400 102400

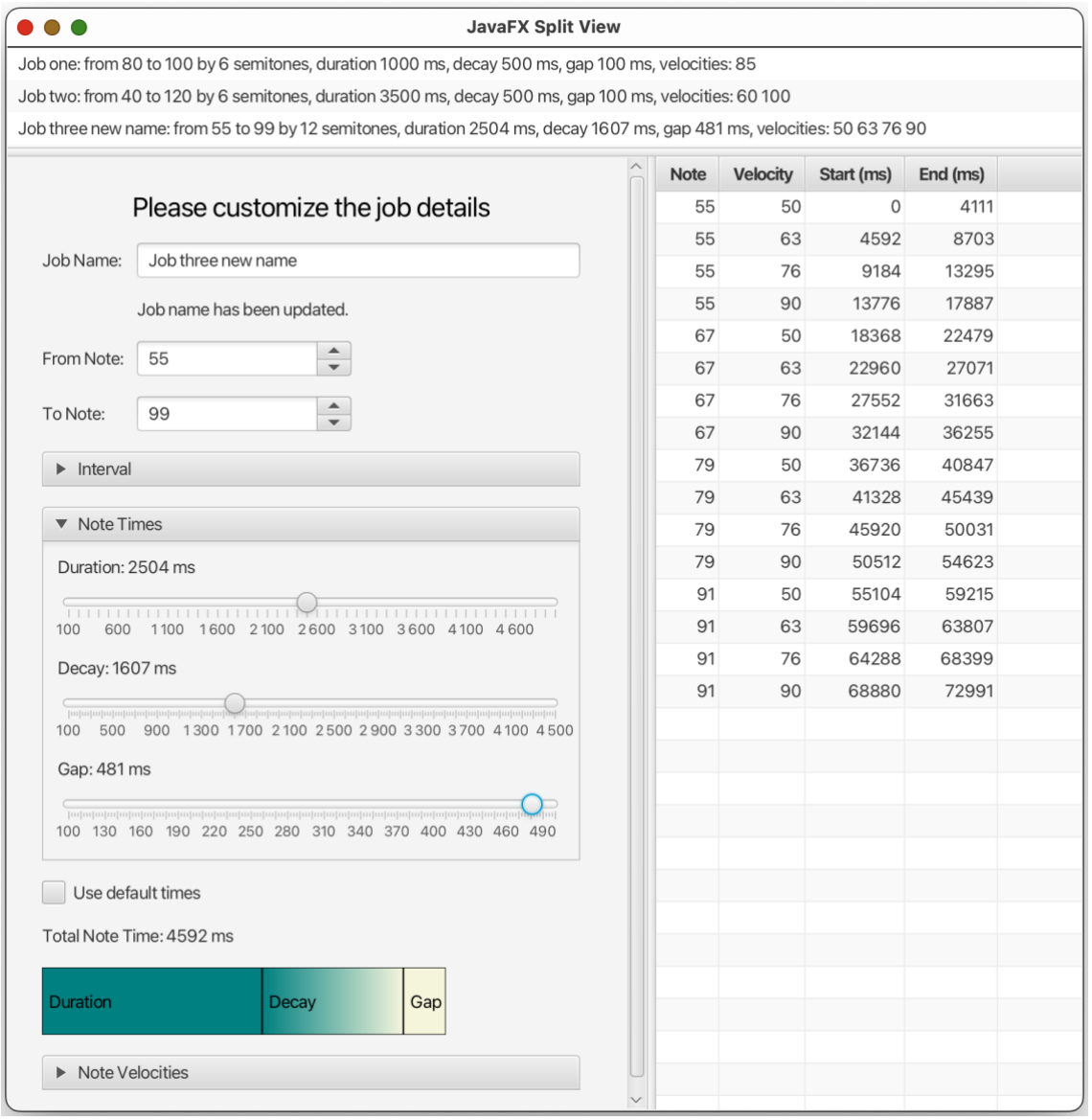


4. When editing the name, from note, to note, or note interval of the current job, both the job list and note table view (first column with note numbers) will be updated.


JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three name too l: from 55 to 99 by 12 semitones, duration 1000 ms, decay 500 ms, gap 300 ms, velocities: 50 63 76 90
Please customize the job details
Job Name:
Job three name too long
Job name should be 1-20 characters long.
From Note:
55
To Note:
99
Interval
ONE
THREE
SIX
TWELVE
Note Times
Use default times
Total Note Time: 1800 ms
Duration
Decay
Gap
Note Velocities
Note
Velocity
Start (ms)
End (ms)
55 50 0 1500
55 63 1800 3300
55 76 3600 5100
55 90 5400 6900
67 50 7200 8700
67 63 9000 10500
67 76 10800 12300
67 90 12600 14100
79 50 14400 15900
79 63 16200 17700
79 76 18000 19500
79 90 19800 21300
91 50 21600 23100
91 63 23400 24900
91 76 25200 26700
91 90 27000 28500

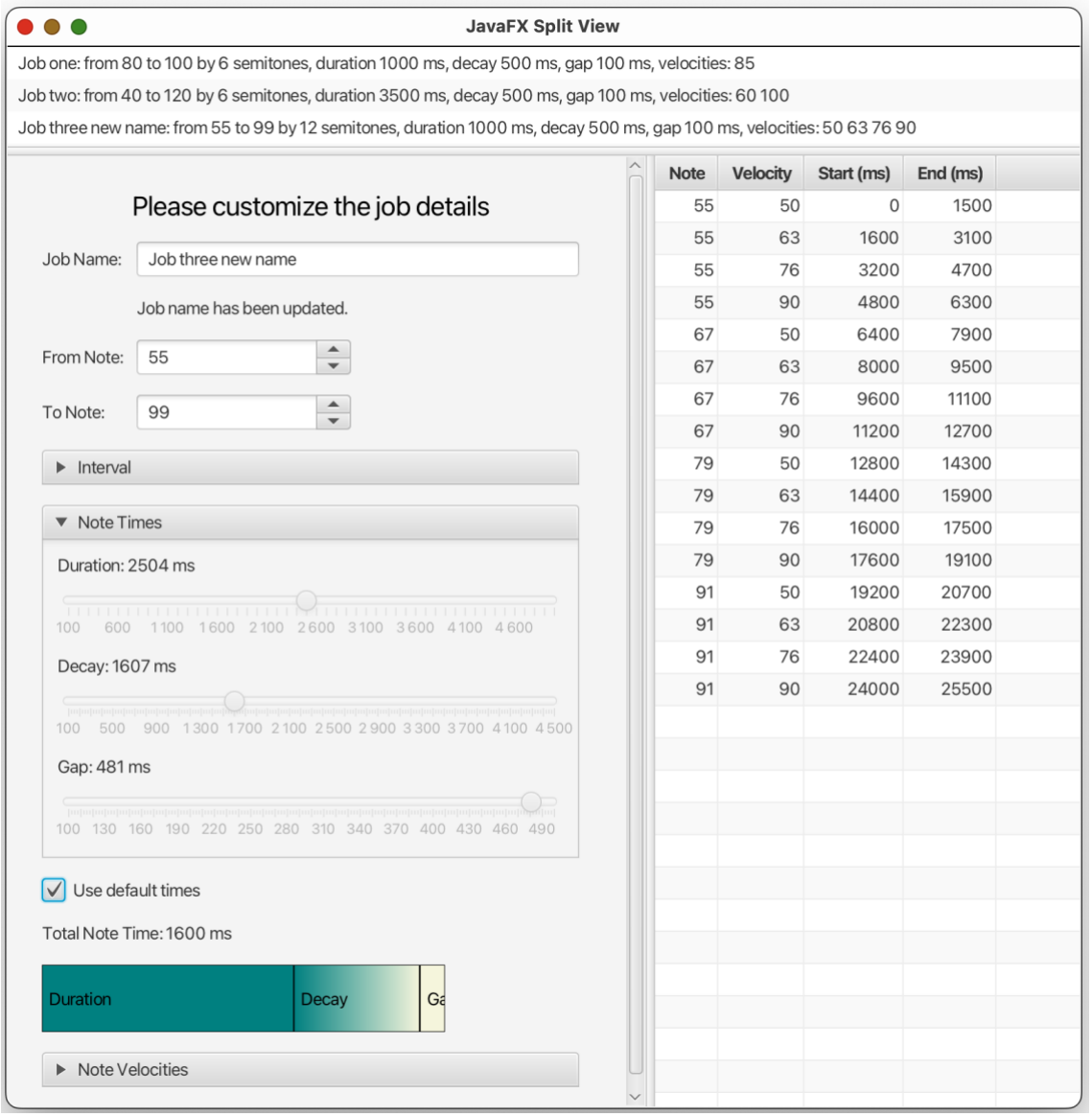


5. When moving the thumbs of the note duration, decay, or gap sliders, the label displaying the total note time and the canvas will be updated, along with the job list and note table view (third column with note start times and fourth column with note end times).

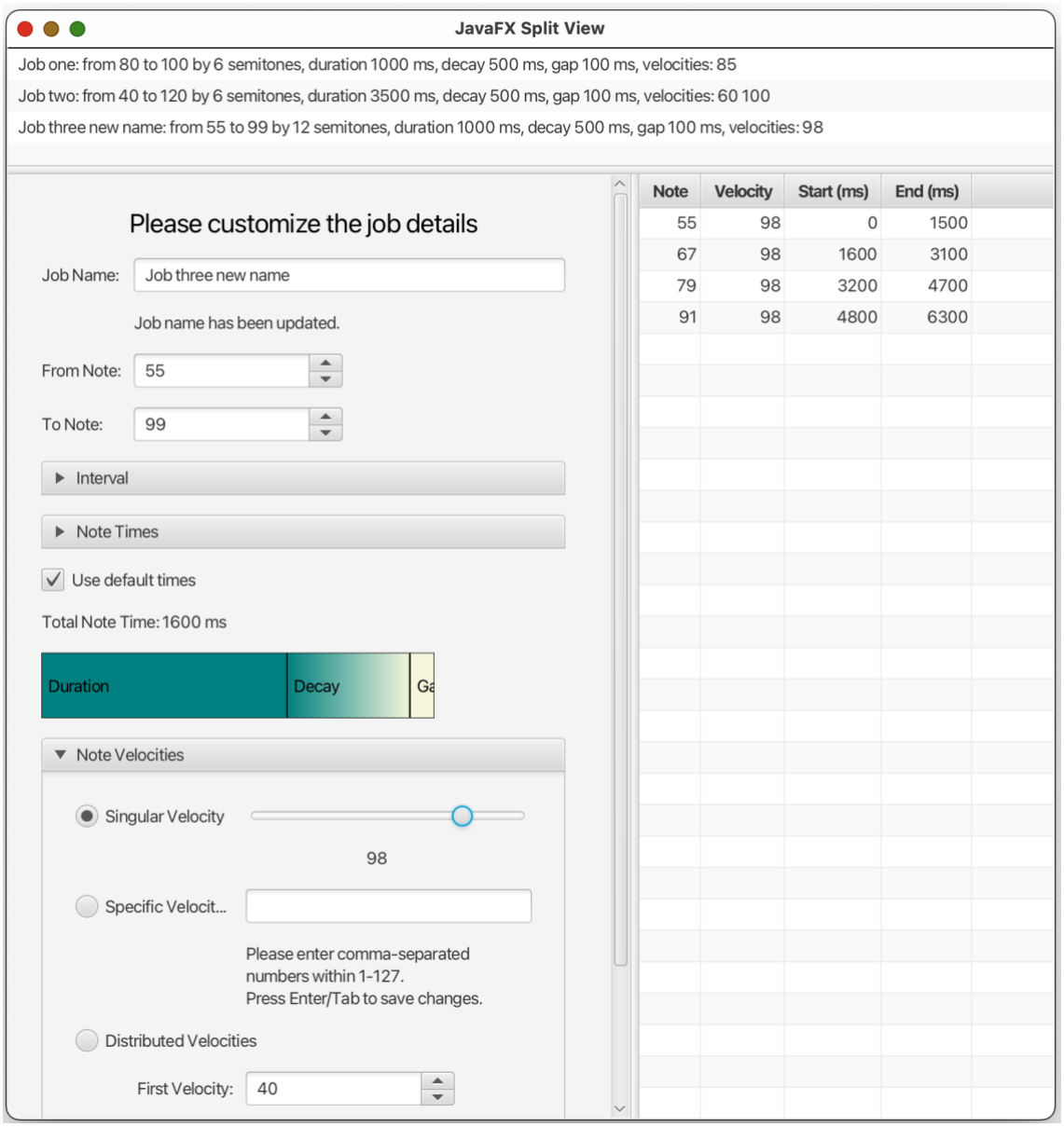


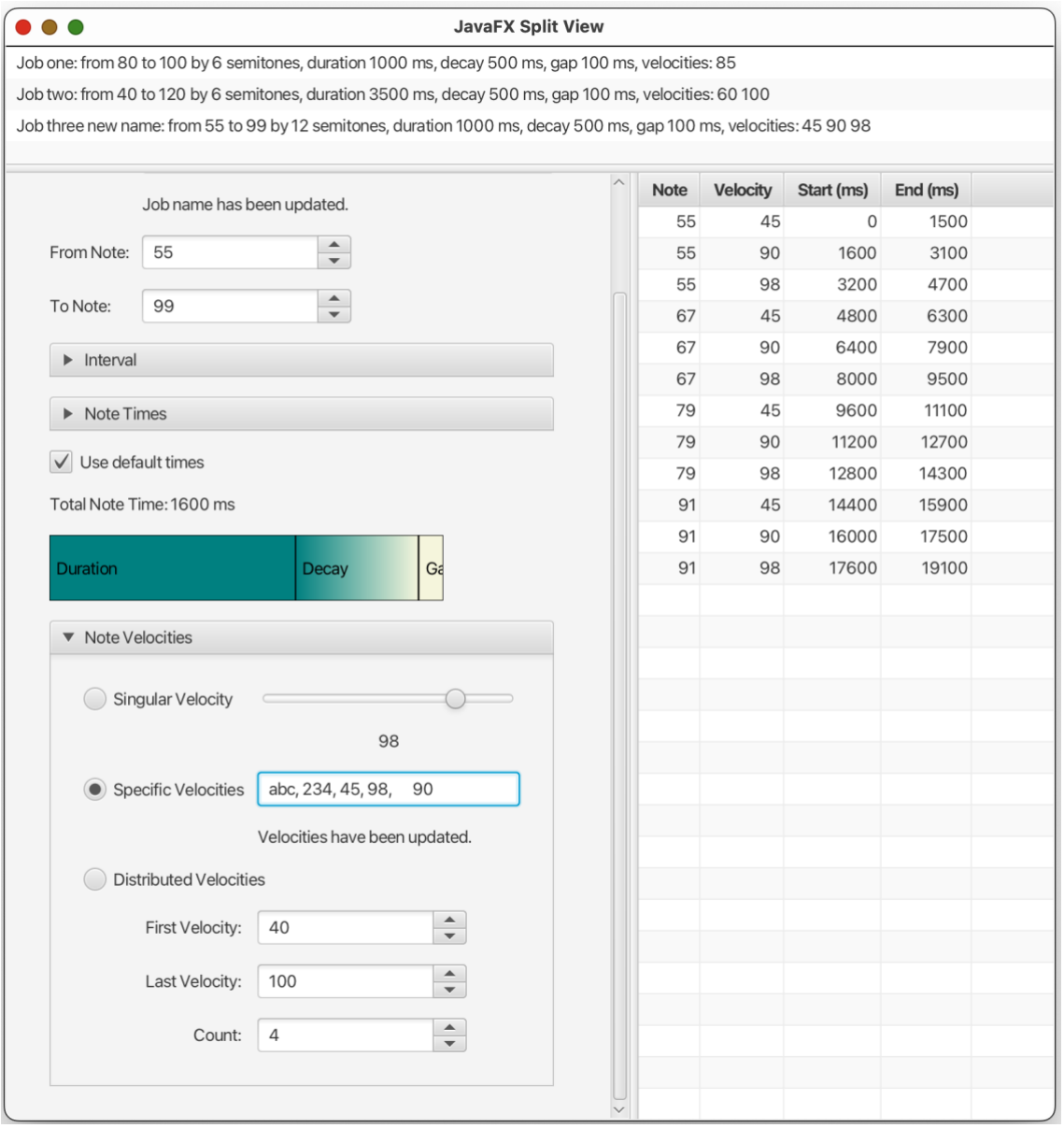
6. Checking the box (Use default times) will result in the current job using default times: duration 1000ms, decay 500ms, gap 100ms. This will update the label displaying total note time and the canvas, as well as the job list and note table view (third column with note start times and fourth

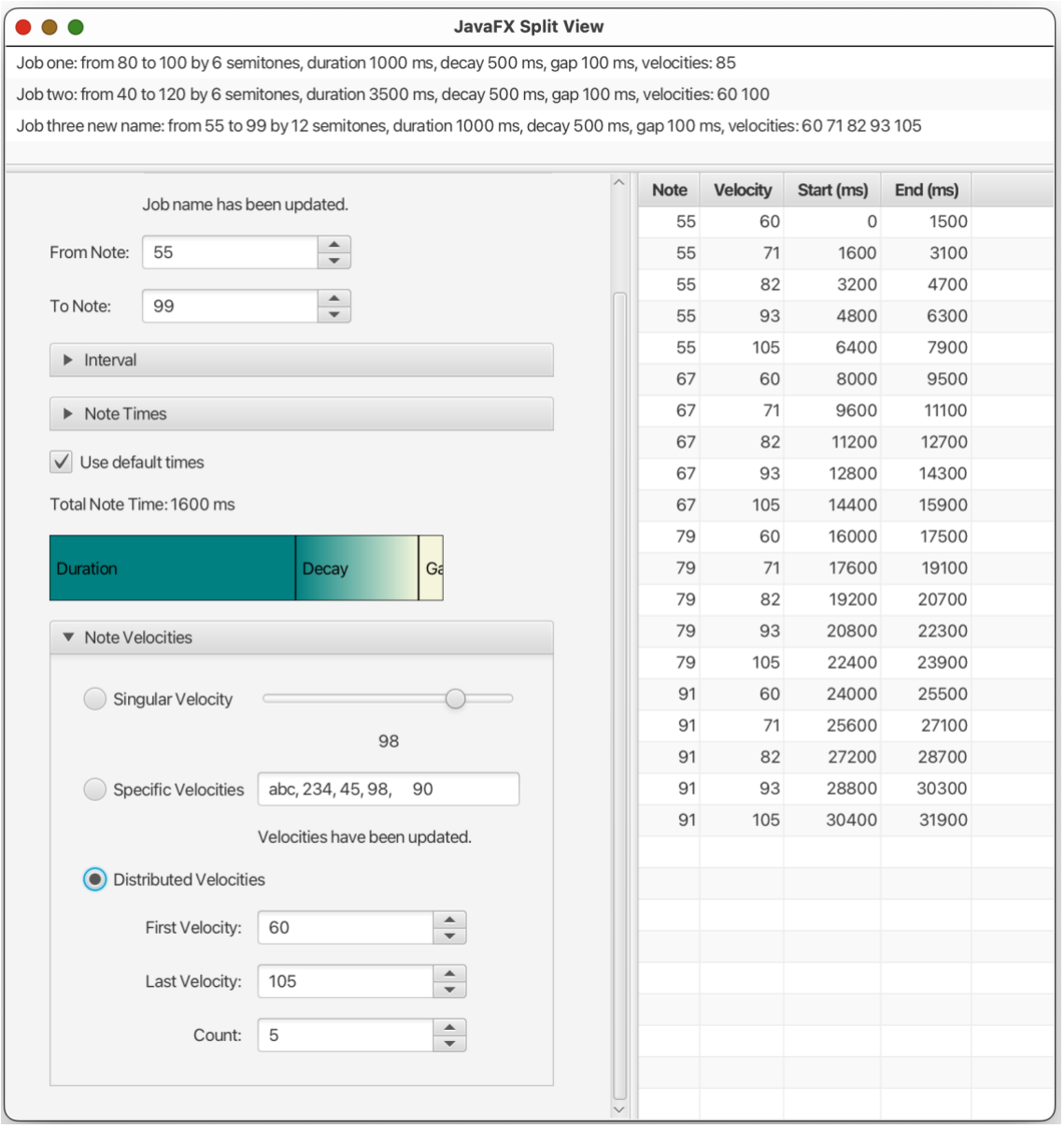
column with note end times).



7. Resetting velocity using singular, specific, or distributed mode will update the job list and note table view (second column with note velocity).


JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three new name: from 55 to 99 by 12 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 98
Please customize the job details
Job Name:
Job three new name
Job name has been updated.
From Note:
55
To Note:
99
Interval
Note Times
Use default times
Total Note Time: 1600 ms
Duration
Decay
Note Velocities
Singular Velocity
98
Specific Velocit...
Please enter comma-separated numbers within 1-127.
Press Enter/Tab to save changes.
Distributed Velocities
First Velocity:
40
Note
Velocity
Start (ms)
End (ms)
55 98 0 1500
67 98 1600 3100
79 98 3200 4700
91 98 4800 6300


JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three new name: from 55 to 99 by 12 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 45 90 98
Job name has been updated.
From Note: 55
To Note: 99
Interval
Note Times
Use default times
Total Note Time: 1600 ms
Duration
Decay
Ga
Note Velocities
Singular Velocity
98
Specific Velocities
abc, 234, 45, 98, 90
Velocities have been updated.
Distributed Velocities
First Velocity: 40
Last Velocity: 100
Count: 4
Note
Velocity
Start (ms)
End (ms)
55 45 0 1500
55 90 1600 3100
55 98 3200 4700
67 45 4800 6300
67 90 6400 7900
67 98 8000 9500
79 45 9600 11100
79 90 11200 12700
79 98 12800 14300
91 45 14400 15900
91 90 16000 17500
91 98 17600 19100


JavaFX Split View
Job one: from 80 to 100 by 6 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 85
Job two: from 40 to 120 by 6 semitones, duration 3500 ms, decay 500 ms, gap 100 ms, velocities: 60 100
Job three new name: from 55 to 99 by 12 semitones, duration 1000 ms, decay 500 ms, gap 100 ms, velocities: 60 71 82 93 105
Job name has been updated.
From Note: 55
To Note: 99
Interval
Note Times
Use default times
Total Note Time: 1600 ms
Duration
Decay
Ga
Note Velocities
Singular Velocity
98
Specific Velocities
abc, 234, 45, 98, 90
Velocities have been updated.
Distributed Velocities
First Velocity: 60
Last Velocity: 105
Count: 5
Note Velocity Start (ms) End (ms)
55 60 0 1500
55 71 1600 3100
55 82 3200 4700
55 93 4800 6300
55 105 6400 7900
67 60 8000 9500
67 71 9600 11100
67 82 11200 12700
67 93 12800 14300
67 105 14400 15900
79 60 16000 17500
79 71 17600 19100
79 82 19200 20700
79 93 20800 22300
79 105 22400 23900
91 60 24000 25500
91 71 25600 27100
91 82 27200 28700
91 93 28800 30300
91 105 30400 31900