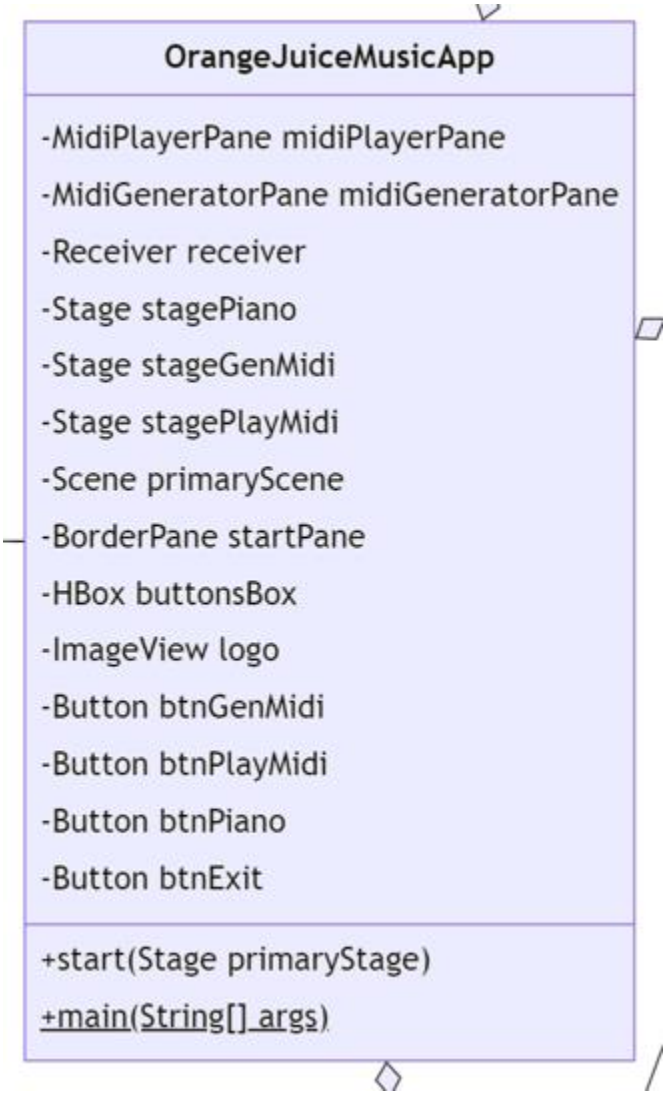
Orange Juice Music App – Implementation Manual

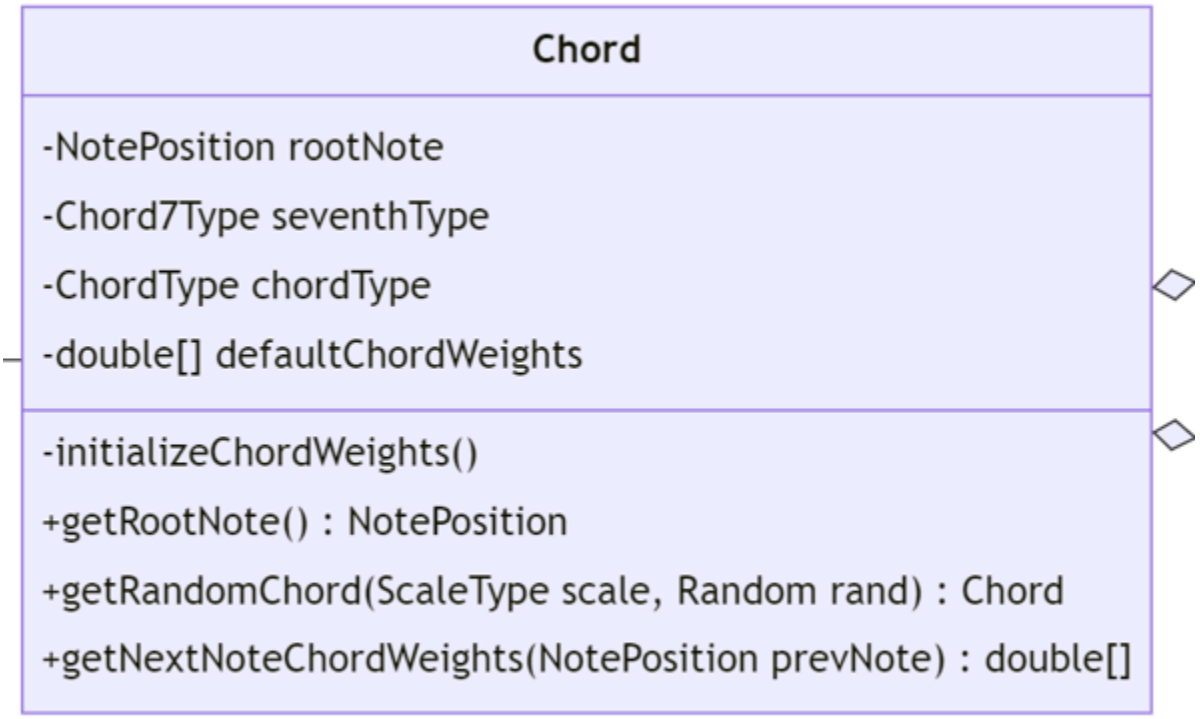
class OrangeJuiceMusicApp



- This class extends the Application class from JavaFX.
- This class contains the main() method and overrides the start() method from the Application class.
- Within the start() method, a few important things are accomplished:
    - The necessary fields are initialized.
    - buttonBox holds the buttons (btnGenMidi, btnPlayMidi, btnPiano, btnExit) and the button actions are set.
        - btnGenMidi shows stageGenMidi
        - btnPlayMidi shows stagePlayMidi
        - btnPiano show stagePiano

- primaryScene is set up to hold the startPane, which gets formatted to look nice and holds the buttonBox and the logo.
- Exit buttons from all panes have their button actions set to close their appropriate stages. (Receivers and Sequencers are closed when necessary.)
- Close requests are handled to close Receivers and Sequencers when necessary and to show the Orange Juice Music Home (primaryStage) again if it has been closed.
- primaryStage holds the primaryScene and is displayed.
- Makes all the stages (except for stagePiano) nonresizable.
- Icons are set for the stages.

class Chord



- initializeChordWeights() – uses 2 switch statements to initialize the defaultChordWeights to a double array of size 12 depending on chordType and seventhType.
- getRandomChord(ScaleType scale, Random rand) – returns a Chord. First generates a random number from 0 to 6 (as there are 7 root notes for chords in the available scale choices in this program, which have a uniform likelihood to appear in this case). Uses nested switch statements to determine the rootNote, chordType, and seventhType depending on the scale. Randomly generates a number from 0 to 3, if that number is not 0, then the seventhType is set to NONE (75% of the time,

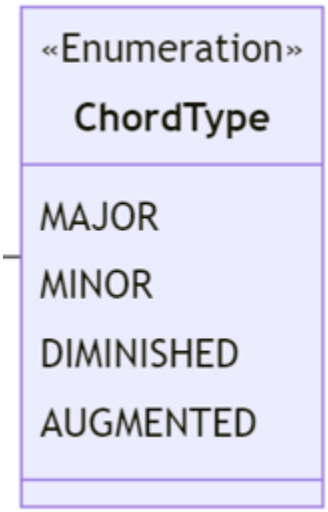
the chord will not be a 7 chord). A new chord is instantiated based upon the determined rootNote, chordType, and seventh type, then it is returned.

- getNextNoteChordWeights(NotePosition prevNote) – returns a double array of size 13. Creates a temporary double array (tempArray) of size 24 and fills it using a for loop depending on the defaultChordWeights with the indices offset based upon the position root note of the chord in the scale. Then creates a new double array (chordWeights) of size 13 and fills it with a for loop with the indices offset the position of the previous note (prevNote) in the scale. Returns chordWeights.
- Class constructor:

```
public Chord(NotePosition rootNote, ChordType chordType, Chord7Type seventhType) {
    this.rootNote = rootNote;
    this.chordType = chordType;
    this.seventhType = seventhType;
    initializeChordWeights();
}
```

- getRootNote() – returns the NotePosition value of the chord's root note (rootNote).

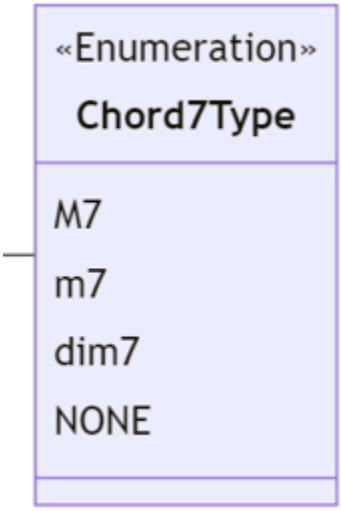
enum ChordType



- An enum:

```
public enum ChordType {
    MAJOR, MINOR, DIMINISHED, AUGMENTED
}
```

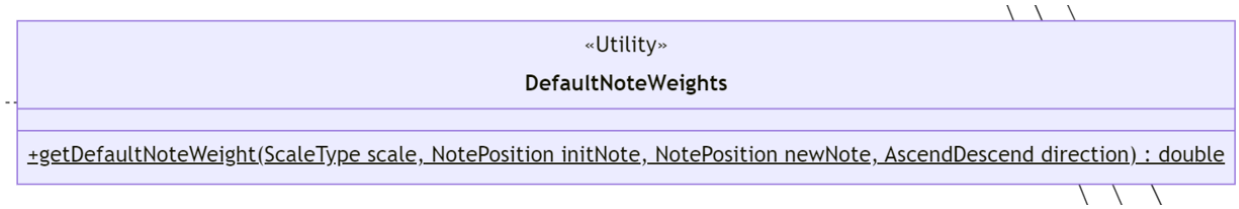
enum Chord7Type



- An enum:

```
public enum Chord7Type {
    M7, m7, dim7, NONE
}
```

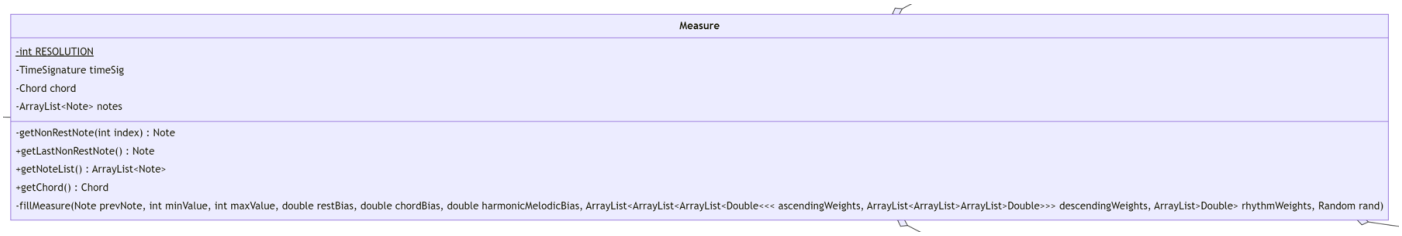
final class DefaultNoteWeights



- A final class with one method. This class was put in a separate file for clarity because there are over 2,200 lines of code in this class (including white spaces, though there are not too many, it is mainly “break;” that takes up space).
- The purpose of this class is to give the initial note weights for each scale type when the user opens the program.
- getDefaultNoteWeight(ScaleType scale, NotePosition initNote, NotePosition newNote, AscendDescend direction) – returns a double representing the weight of the note for when the song is randomly generated. The method consists of nested switch statements. The outermost is “switch (direction)”, then “switch (scale)”, then “switch (initNote)”, then finally “switch (newNote)”. (initNote stands for “initial note”, it represents the value in the “Starting Note:” combo-box (cboInitialNote) on the ScaleWeightsPane in the MidiGeneratorPane. newNote

represents the NotePosition value associated with each Spinner on a NoteWeightPane. The default weights will not be listed here due to length (there are 4 ScaleTypes, 12 NotePositions for initNote, 13 NotePositions for newNote, and either ASCEND or DESCEND for directions, which is 4*12*13*2 = 1248 Spinners which need initial values (though a fair portion are 0 which can be handled with a default case.))

class Measure



- RESOLUTION is a final int and is 64 (ticks per quarter note).
- getNonRestNote(int index) – returns a Note (may be null). This private method recursively searches the measure (namely the ArrayList<Note> notes) for the final note which is not a rest. If the method reaches the base case (index is 0) and a note which is not a rest is still not found, null is returned, otherwise the method returns the first non-rest note that it finds.
- getLastNonRestNote() – returns a Note (may be null). This public method calls the private method "getNonRestNote(int index)" with the input index being the last index of ArrayList<Note> notes. The method returns the final non-rest note in the measure if any, otherwise it returns null.
- getNoteList() – returns ArrayList<Note> (notes).
- getChord() – returns Chord (chord).
- fillMeasure(Note prevNote, int minValue, int maxValue, double restBias, double chordBias, double harmonicMelodicBias, ArrayList<ArrayList<ArrayList<Double>>> ascendingWeights, ArrayList<ArrayList<ArrayList<Double>>> descendingWeights, ArrayList<Double> rhythmWeights, Random rand) – Most important method in this class! Creates a placeholder Note (lastNonRestNote) and initializes sets it as prevNote. Creates an int (remainingTicks) and determines its value based upon the TimeSignature timeSig and RESOLUTION. Uses a while loop (while (remainingTicks > 0)) to fill the measure as follows:
    - Determine if the next note is a rest.
    - Determine the next note's value with the Note class's getNextNoteValue() method, using lastNonRestNote as the previous note.
    - Determine the next note's length type with the Note class's getNextNoteLengthType() method.
    - Create a Note (nextNote) and add it to the ArrayList<Note> notes.
    - Subtract the next note's length value from remainingTicks.
    - If nextNote is not a rest, set lastNonRestNote as nextNote before looping.

- Class constructor:

```java
public Measure(Note prevNote, TimeSignature timeSig, Chord chord, int minValue, int maxValue, double restBias,
        double chordBias,
        double harmonicMelodicBias, ArrayList<ArrayList<ArrayList<Double>>> ascendingWeights,
        ArrayList<ArrayList<ArrayList<Double>>> descendingWeights, ArrayList<Double> rhythmWeights, Random rand) {
    this.chord = chord;
    this.timeSig = timeSig;
    this.notes = new ArrayList<Note>();
    fillMeasure(prevNote, minValue, maxValue, restBias, chordBias, harmonicMelodicBias, ascendingWeights,
            descendingWeights, rhythmWeights, rand);
}
```

class MidiGeneratorPane

- This class extends the BorderPane class from JavaFX.
- ObservableList fields for ComboBox fields are as follows:

```
private static ObservableList<String> scalesList = FXCollections.observableArrayList("Major", "Natural Minor",
        "Harmonic Minor", "Melodic Minor");
private static ObservableList<String> noteList = FXCollections.observableArrayList("A", "A#/Bb", "B", "C", "C#/Db",
        "D", "D#/Eb", "E", "F", "F#/Gb", "G", "G#/Ab");
```

- playMidi() – Plays currentSequence with sequencer.
- pauseMidi() – Stops the sequencer.
- stopMidi() – Stops the sequencer and sets the sequencer tick position to 0.
- getAscendScaleWeights(ScaleType scale) – Returns an ArrayList<ArrayList<Double>>. The outer ArrayList has indices from 0 to 11 (corresponding to the 12 choices for "Starting Note" (cboInitialNote) on the ScaleWeightsPane, and the inner ArrayList has indices from 0 to 12 (corresponding to the 13 "Ascend" Spinners on an "Ascend" NoteWeightsPane). Uses a switch statement for ScaleType scale.
- getDescendScaleWeights(ScaleType scale) – Returns an ArrayList<ArrayList<Double>>. The outer ArrayList has indices from 0 to 11 (corresponding to the 12 choices for "Starting Note" (cboInitialNote) on the ScaleWeightsPane, and the inner ArrayList has indices from 0 to 12 (corresponding to the 13 "Ascend" Spinners on an "Ascend" NoteWeightsPane). Uses a switch statement for ScaleType scale.
- getSequencer() – Returns Sequencer (sequencer) so it can be closed by other classes if necessary.
- getExitButton() – Returns Button (btnExit) so that the class OrangeJuiceMusicApp can set the button's action to close the stage.
- Class Constructor:

```
public MidiGeneratorPane() {
    super();
```

  - The necessary fields are initialized.
  - cboScaleType's action is set to change the weights displayed in the center of the pane depending upon the selected scale type.
  - btnPlay, btnStop, and btnSave have their visibility set to false.
  - btnGen's action is set to do the following:
    - Stop and close the sequencer (if necessary).
    - Make a new instance of the sequencer.
    - Read in the necessary data from all the nodes on the pane. (The melody and bass ranges are adjusted based upon the scale root note such that the song generates as if the root note was C).
    - Set currentSong as a new instance of a song with the data read in from the pane.
    - Set the seed of currentSong with the seed from the TextField txtRandomSeed.
    - Use the generateSong() method for currentSong.
    - Set currentSequence as currentSong.convertToSequence().

        - Set btnPlay's text to "Play"
        - Make btnPlay, btnStop, btnSave become visible.
    - btnPlay's action is set to switch text to "Pause" and call the playMidi() method or switch text to "Play" and call the pauseMidi() method depending upon the current text of the button.
    - btnStop's action is set to set btnPlay's text to "Play" and call the stopMidi() method.
    - btnSave's action is set to show a FileChooser to allow the user to save currentSequence as a midi file.
    - The pane is formatted to look nice.

class MidiPlayerPane

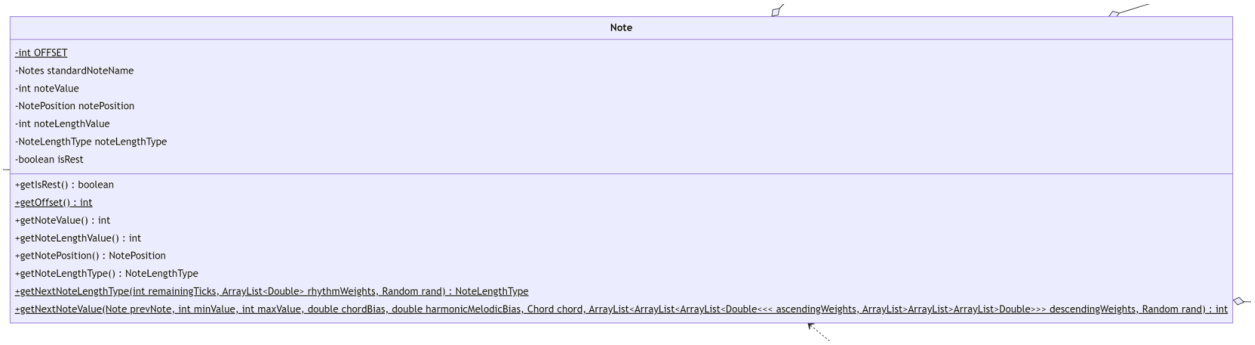
- This class extends the BorderPane class from JavaFX.
- closeTimer() – Closes the Timer (timer). This method is typically called when the stage holding the MidiPlayerPane is closed.
- getExitButton() – Returns Button (btnExit) so that the class OrangeJuiceMusicApp can set the button's action to close the stage.
- getSequencer() – Returns Sequencer (sequencer) so it can be closed by other classes if necessary.
- loadMidi() – The currently playing midi file will be stopped (if applicable) and the Sequencer (sequencer) will be closed. A FileChooser is then used to allow the user to select the midi file that they would like to play. In a try block, a new Sequencer instance is created as sequencer, and the previously selected midi file is turned into a Sequence (sequence) using static method MidiSystem.getSequence(File file) (MidiSystem is from the javax.sound.midi package). The Label (lblFileName)'s text is then set to the midi file's name and btnPlayMidi and btnStopMidi are made visible. If something goes wrong, the catch block makes lblFileName display the text "Failed to load .mid file".
- playMidi() – Opens the Sequencer (sequencer) if it is not already open. Sets the Sequence as the current sequence (sequence), then starts the sequencer. Finally, the boolean (timerIsPaused) is set to false.
- pauseMidi() – Stops the Sequencer (sequencer). Sets timerIsPaused to true.
- stopMidi() – Stops the Sequencer (sequencer). Sets timerIsPaused to true. Sets timerTimeMilliseconds to 0 and resets txtTimer's text to "00:00:00". Sets the tick position of sequencer back to 0.
- Class Constructor:

```
public MidiPlayerPane() {
    super();
```

  - The necessary fields are initialized.
  - The actions for btnLoadMidi, btnPlayMidi, and btnStopMidi are set:
    - btnLoadMidi – Sets btnPlayMidi's text to "Play" and calls loadMidi().
    - btnPlayMidi – Sets btnPlayMidi's text to "Pause" and calls playMidi() if the current text is "Play". Sets btnPlayMidi's text to "Play" and calls pauseMidi() if the current text is "Pause".
    - btnStopMidi – Set btnPlayMidi's text to "Play" and calls stopMidi().
  - An instance of TimerTask is created for timerTask which has a run() method that does the following:
    - Checks if the timer is not paused (with the boolean timerIsPaused).
    - If it is not paused, timerTimeMilliseconds is incremented by TIMER_MILLISECOND_UPDATE and timerUpdateCounter is incremented by 1.
    - If one second has passed based upon the timerUpdateCounter, txtTimer's text is updated to show the current elapsed time "(hours):(minutes):(seconds)".

- An instance of Timer is created for timer, and the TimerTask (timerTask) is scheduled at a fixed rate with no delay and a period of TIMER_MILLISECOND_UPDATE.
- btnPlayMidi and btnStopMidi have their visibility set to false.
- The ImageView (logo) is set to the center of the pane after having loaded in the logo image as an Image to use in the constructor for ImageView.
- The pane is formatted to look nice.

class Note



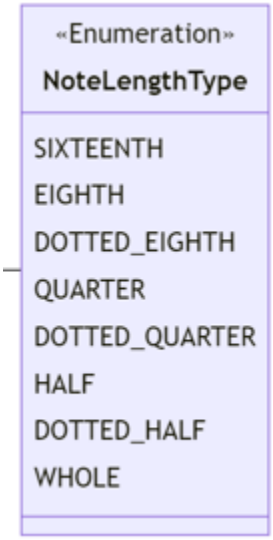
- OFFSET is a final int with a value of 12.
- standardNoteName is the name of the note if it the scale's root note was C. (When music is being generated, the settings are converted to having C as the scale's root note, then when converted to a Sequence, the values are adjusted to be back to match the originally selected root note.)
- getIsRest(), getOffset(), getNoteValue(), getNoteLengthValue(), getNotePosition(), getNoteLengthType() are simple getter methods.
- getNextNoteLengthType(int remainingTicks, ArrayList<Double> rhythmWeights, Random rand) – Returns a NoteLengthType. This method is a very important method for the midi generation. The weights are first read in from the ArrayList<Double> (rhythmWeights) and each stored in a separate double p1 (whole note probability), p2 (dotted half note probability), …, p8 (sixteenth note probability). Depending on if remainingTicks is greater than or equal to the note length value (in ticks) for a given NoteLengthType (based upon NoteUtil's noteTickLength64ResolutionMap), a double[] (weights) is created (which contains the probabilities for only those NoteLengthTypes which can fit in the remaining number of ticks). A weighted random is got from weights by using NoteUtil's getWeightedRandom() method. Depending upon the index which that method returns, the corresponding NoteLengthType is returned from this method.
- getNextNoteValue(Note prevNote, int minValue, int maxValue, double chordBias, double harmonicMelodicBias, Chord chord, ArrayList<ArrayList<ArrayList<Double>>> ascendingWeights, ArrayList<ArrayList<ArrayList<Double>>> descendingWeights, Random rand) – Returns an int (representing the midi pitch value for the next note). This method is a very important method for the midi generation. First, it is determined whether

this note will ascend or descend based upon NoteUtil's getAscendDescend() method. Then, the corresponding ascending/descending weights are obtained from their respective ArrayList and stored in a double[] of size 13 (noteWeights). The outermost ArrayList has 2 indices, which correspond to the two scale types (which will be either both the same, or natural and harmonic minor or natural and melodic minor if the selected scale type is harmonic or melodic minor), and depending on harmonicMelodicBias, one of these is selected. The next nested ArrayList refers to the NotePosition of prevNote, and the desired index is obtained from NoteUtil's notesAscendValueMap. The innermost ArrayList holds Doubles and is used to fill noteWeights. The chord weights are then gotten from the Chord's getNextNoteChordWeights() method and stored in a double[] (chordWeights). Next, noteWeights and chordWeights are normalized (to sum to 1) using NoteUtil's normalizeWeights() method. A double[] (finalWeights) is created and its elements are the sum of the corresponding elements of noteWeights and chordWeights summed together after each being multiplied a value determining their weight relative to each other (this value is based upon chordBias). A weighted random integer is got using NoteUtil's getWeightedRandom() method, and this value is added to the previous note's value if ascending, or added to the previous note's value and then has 12 subtracted if descending. If the note value is less than minValue (for the note range), 12 is added to it (an octave). If the note value is greater than maxValue (for the note range), 12 is subtracted from it (an octave).

- Class Constructors:
    - This class has two constructors, one is public and the other is protected.
    - The protected constructor is used to make a "pseudo-Note" because the note generation relies upon a previous (non-rest) note to generate the next note, this constructor gets used to make a note that has a pitch value, but no length. In this program, this constructor is used with the pitch value being the midpoint of the pitch range as an integer value, so that generation begins in the middle of the note range. This "pseudo-Note" does not get added to any Measures.
    - The public constructor is the constructor used to make a Note which gets added to a Measure.

```
protected Note(int noteValue) {

    this.noteValue = noteValue;

    this.standardNoteName = NoteUtil.calcNoteName(noteValue);
    this.notePosition = NoteUtil.getNotesNameToStandardPositionMap().get(standardNoteName);
    this.isRest = false;
    this.noteLengthValue = 0;
    this.noteLengthType = null;
}

public Note(int noteValue, NoteLengthType noteLength, boolean rest) {

    this.isRest = rest;
    this.noteValue = noteValue;
    this.noteLengthType = noteLength;
    this.noteLengthValue = NoteUtil.getNoteTickLength64ResolutionMap().get(noteLength);
    this.standardNoteName = NoteUtil.calcNoteName(noteValue);
    this.notePosition = NoteUtil.getNotesNameToStandardPositionMap().get(standardNoteName);
}
```

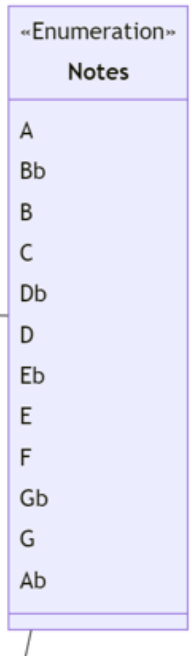
enum NoteLengthType



- An enum:

```
public enum NoteLengthType {
    SIXTEENTH, EIGHTH, DOTTED_EIGHTH, QUARTER, DOTTED_QUARTER, HALF, DOTTED_HALF, WHOLE
}
```

enum Notes



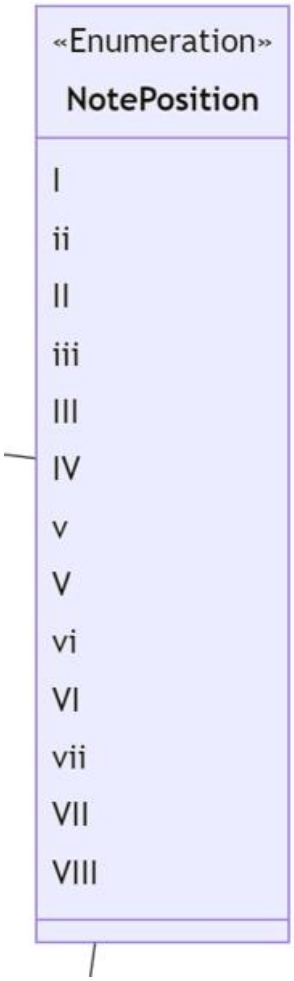
- Do not confuse “Notes” with “Note”. Note is a class, Notes is an enum. Notes lists all the possible note names (only the flat names are listed for sharps/flats, so instead of saying F#, Gb is used), of which there are 12.
- An enum:

```
public enum Notes {
    A, Bb, B, C, Db, D, Eb, E, F, Gb, G, Ab
}
```

class NoteUtil

- A final class with only static fields.
- This class has two enums as inner classes:

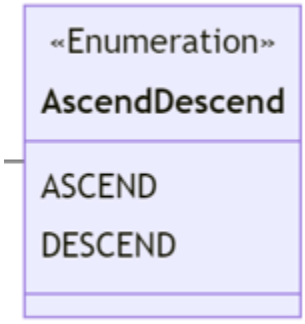
- enum NotePosition



  - Represents relative note position (see the Maps in the NoteUtil class for a better understanding).
  - An enum:

```
public static enum NotePosition {
    I, ii, II, iii, III, IV, v, V, vi, VI, vii, VII, VIII
}
```

- enum AscendDecend



        - This enum could have been a boolean, but was done this way for a bit of clarity.
        - An enum:

```
public static enum AscendDescend {
    ASCEND, DESCEND
}
```

- getAscendDescend(int prevNoteValue, int minValue, int maxValue, Random rand) – Returns ASCEND or DESCEND. This method decides whether the next note should ascend or descend based upon a normal distribution curve. The mean of the normal distribution is estimated as the average of maxValue and minValue (which gets stored as a double (midValue) in this method). The standard deviation is estimated as a third of the distance between maxValue and midValue. A random value is gotten from the Random class's nextGaussian() method based upon this mean and standard deviation. Depending on how this random value compares which the value of the previous note (prevNoteValue), ASCEND or DESCEND is returned. In general, if prevNoteValue is less than midValue, then the method will most likely return ASCEND, and vice versa for DESCEND.
- normalizeWeights(double[] weights, int weightsSize) – This method modifies the input array so that its elements sum to 1. If the original sum of the elements is 0, 1 is stored in the 0th index of the array and returned to prevent a divide by 0 error.
- getWeightedRandom(double[] weights, int weightsSize, Random rand) – This method returns an int corresponding to the index of the randomly selected weight. The weights are first normalized using the normalizeWeights() method. Next, a random double between 0 and 1 is generated. Then a for loop is used to iterate through the list to determine which weight range the random double falls within, using two values to store sums for comparison. The index for the weight which is chosen is returned. (Larger weight values mean a larger range for the random double to fall within, and thus are more likely to have their index returned.)

- notesAscendValueMap – A map as follows:

```
private static Map<NotePosition, Integer> notesAscendValueMap = Map.ofEntries(
        entry(NotePosition.I, v: 0),
        entry(NotePosition.ii, v: 1),
        entry(NotePosition.II, v: 2),
        entry(NotePosition.iii, v: 3),
        entry(NotePosition.III, v: 4),
        entry(NotePosition.IV, v: 5),
        entry(NotePosition.v, v: 6),
        entry(NotePosition.V, v: 7),
        entry(NotePosition.vi, v: 8),
        entry(NotePosition.VI, v: 9),
        entry(NotePosition.vii, v: 10),
        entry(NotePosition.VII, v: 11),
        entry(NotePosition.VIII, v: 12));
```

- notesDescendValueMap – A map as follows:

```
private static Map<NotePosition, Integer> notesDescendValueMap = Map.ofEntries(
        entry(NotePosition.VIII, v: 0),
        entry(NotePosition.VII, -1),
        entry(NotePosition.vii, -2),
        entry(NotePosition.VI, -3),
        entry(NotePosition.vi, -4),
        entry(NotePosition.V, -5),
        entry(NotePosition.v, -6),
        entry(NotePosition.IV, -7),
        entry(NotePosition.III, -8),
        entry(NotePosition.iii, -9),
        entry(NotePosition.II, -10),
        entry(NotePosition.ii, -11),
        entry(NotePosition.I, -12));
```

- noteMap – A map as follows:

```
private static Map<Notes, Integer> noteMap = Map.ofEntries(
        entry(Notes.C, v: 0),
        entry(Notes.Db, v: 1),
        entry(Notes.D, v: 2),
        entry(Notes.Eb, v: 3),
        entry(Notes.E, v: 4),
        entry(Notes.F, v: 5),
        entry(Notes.Gb, v: 6),
        entry(Notes.G, v: 7),
        entry(Notes.Ab, v: 8),
        entry(Notes.A, v: 9),
        entry(Notes.Bb, v: 10),
        entry(Notes.B, v: 11));
```

- scaleOffsetMap – A map (determining the offset of the chosen root note from C and is used when generating music) which is as follows:

```
private static Map<Notes, Integer> scaleOffsetMap = Map.ofEntries(
        entry(Notes.C, v: 0),
        entry(Notes.Db, v: 1),
        entry(Notes.D, v: 2),
        entry(Notes.Eb, v: 3),
        entry(Notes.E, v: 4),
        entry(Notes.F, v: 5),
        entry(Notes.Gb, v: 6),
        entry(Notes.G, v: 7),
        entry(Notes.Ab, v: 8),
        entry(Notes.A, -3),
        entry(Notes.Bb, -2),
        entry(Notes.B, -1));
```

- noteTickLength64ResolutionMap – A map as follows:

```
private static Map<NoteLengthType, Integer> noteTickLength64ResolutionMap = Map.ofEntries(
        entry(NoteLengthType.SIXTEENTH, v: 16),
        entry(NoteLengthType.EIGHTH, v: 32),
        entry(NoteLengthType.DOTTED_EIGHTH, v: 48),
        entry(NoteLengthType.QUARTER, v: 64),
        entry(NoteLengthType.DOTTED_QUARTER, v: 96),
        entry(NoteLengthType.HALF, v: 128),
        entry(NoteLengthType.DOTTED_HALF, v: 192),
        entry(NoteLengthType.WHOLE, v: 256));
```

- notesStandardPositionToNameMap – Standard position means it is relative to a root note of C. A map as follows:

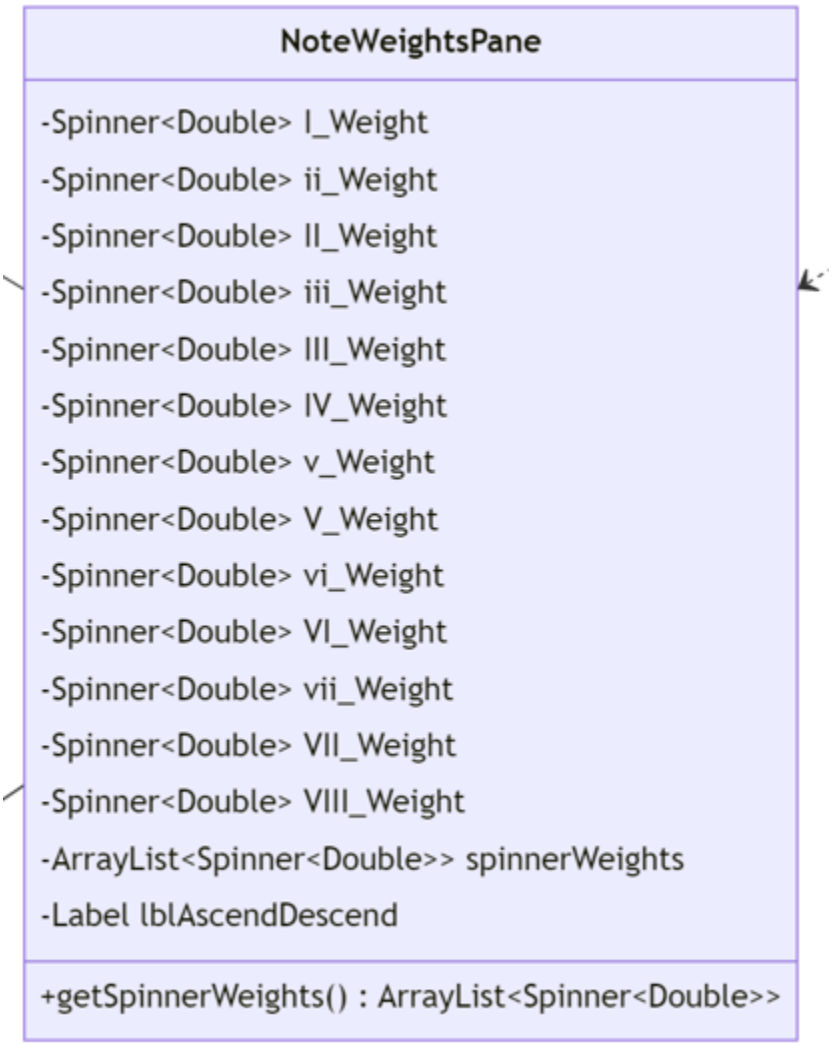
```
private static Map<NotePosition, Notes> notesStandardPositionToNameMap = Map.ofEntries(
        entry(NotePosition.I, Notes.C),
        entry(NotePosition.ii, Notes.Db),
        entry(NotePosition.II, Notes.D),
        entry(NotePosition.iii, Notes.Eb),
        entry(NotePosition.III, Notes.E),
        entry(NotePosition.IV, Notes.F),
        entry(NotePosition.v, Notes.Gb),
        entry(NotePosition.V, Notes.G),
        entry(NotePosition.vi, Notes.Ab),
        entry(NotePosition.VI, Notes.A),
        entry(NotePosition.vii, Notes.Bb),
        entry(NotePosition.VII, Notes.B),
        entry(NotePosition.VIII, Notes.C));
```

- notesNameToStandardPositionMap - Standard position means it is relative to a root note of C. A map as follows:

```
private static Map<Notes, NotePosition> notesNameToStandardPositionMap = Map.ofEntries(
        entry(Notes.C, NotePosition.I),
        entry(Notes.Db, NotePosition.ii),
        entry(Notes.D, NotePosition.II),
        entry(Notes.Eb, NotePosition.iii),
        entry(Notes.E, NotePosition.III),
        entry(Notes.F, NotePosition.IV),
        entry(Notes.Gb, NotePosition.v),
        entry(Notes.G, NotePosition.V),
        entry(Notes.Ab, NotePosition.vi),
        entry(Notes.A, NotePosition.VI),
        entry(Notes.Bb, NotePosition.vii),
        entry(Notes.B, NotePosition.VII));
```

- getNoteValue(Notes note, int octave) – Returns an int representing the midi value for the note with the given note name and octave. Makes use of the noteMap.
- calcNoteName(int noteValue) – Returns a Notes (the note name) for a given midi pitch value for a note.
- As all the Maps for this class are private, the class provides public getter methods for each Map.

class NoteWeightsPane

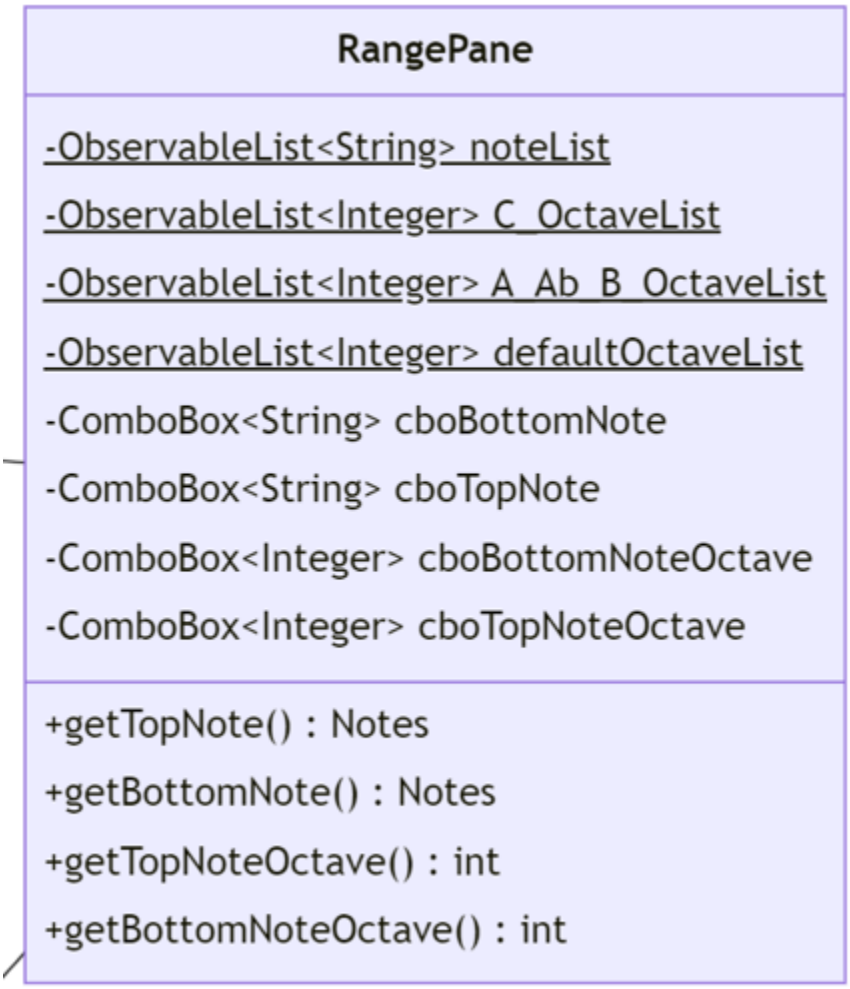


- This class extends the GridPane class from JavaFX.
- getSpinnerWeights() – Returns an ArrayList<Spinner<Double>> (spinnerWeights).
- Class Constructor:

```
public NoteWeightsPane(ScaleType scale, NotePosition initNote, AscendDescend direction) {
    super();
```

  - The necessary fields are initialized.
  - Spinners are created with their default value gotten from the getDefaultNoteWeight() method from the DefaultNoteWeights class, and then added to the ArrayList<Spinner<Double>> (spinnerWeights).
  - Using a switch statement for direction, the Spinners are then added to the pane with Labels and the pane is formatted to look nice.

class RangePane



- This class extends the VBox class from JavaFX.
- The 4 ObservableLists are as follows:

```
private static ObservableList<String> noteList = FXCollections.observableArrayList("A", "A#/Bb", "B", "C", "C#/Db",
        "D", "D#/Eb", "E", "F", "F#/Gb", "G", "G#/Ab");
private static ObservableList<Integer> C_OctaveList = FXCollections.observableArrayList(1, 2, 3, 4, 5, 6, 7, 8);
private static ObservableList<Integer> A_Ab_B_OctaveList = FXCollections.observableArrayList(0, 1, 2, 3, 4, 5, 6,
        7);
private static ObservableList<Integer> defaultOctaveList = FXCollections.observableArrayList(1, 2, 3, 4, 5, 6, 7);
```

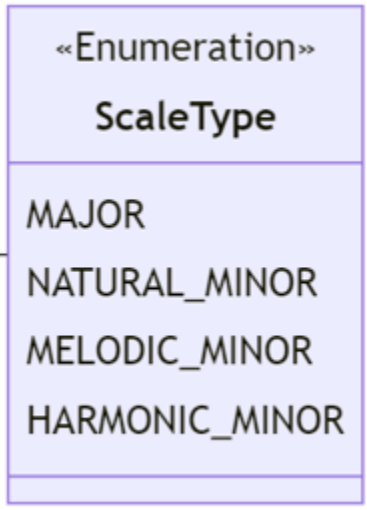
- getTopNote() – Returns a Notes value corresponding to the currently selected value in cboTopNote.
- getBottomNote() – Returns a Notes value corresponding to the currently selected value in cboBottomNote.
- getTopNoteOctave() – Returns an int value corresponding to the currently selected value in cboTopNoteOctave.
- getBottomNoteOctave() – Returns an int value corresponding to the currently selected value in cboBottomNoteOctave.

- Class Constructor:

```
public RangePane(String partName) {
    super(5);
```

  - The necessary fields are initialized.
  - Sets actions for the ComboBoxes to display the appropriate values and never display a value which is out of the range of notes on a piano (A0 to C8).
  - The pane is formatted to look nice.

enum ScaleType



- An enum:

```
public enum ScaleType {
    MAJOR, NATURAL_MINOR, MELODIC_MINOR, HARMONIC_MINOR
}
```

class ScaleWeightsPane

- This class extends the BorderPane class from JavaFX.
- The ObservableList<String> (initNoteList) for the ComboBox<String> (cboInitialNote) is as follows:

```
private static ObservableList<String> initNoteList = FXCollections.observableArrayList("I", "ii", "II", "iii",
        "III", "IV", "v", "V", "vi", "VI", "vii", "VII");
```

- getAscendingWeights() – Returns an ArrayList<NoteWeightsPane> (ascendingNoteWeights).
- getDescendingWeights() – Returns an ArrayList<NoteWeightsPane> (descendingNoteWeights).
- Class Constructor:

```
public ScaleWeightsPane(ScaleType scale) {
    super();
```

  - The necessary fields are initialized.
  - Ascending weights are added to ascendingWeights in order (I to VII).
  - Descending weights are added to descendingWeights in order (I to VII).
  - The action for cboInitialNote is set to display the proper set of ascending and descending weights.
  - The pane is formatted to look nice.

class Song

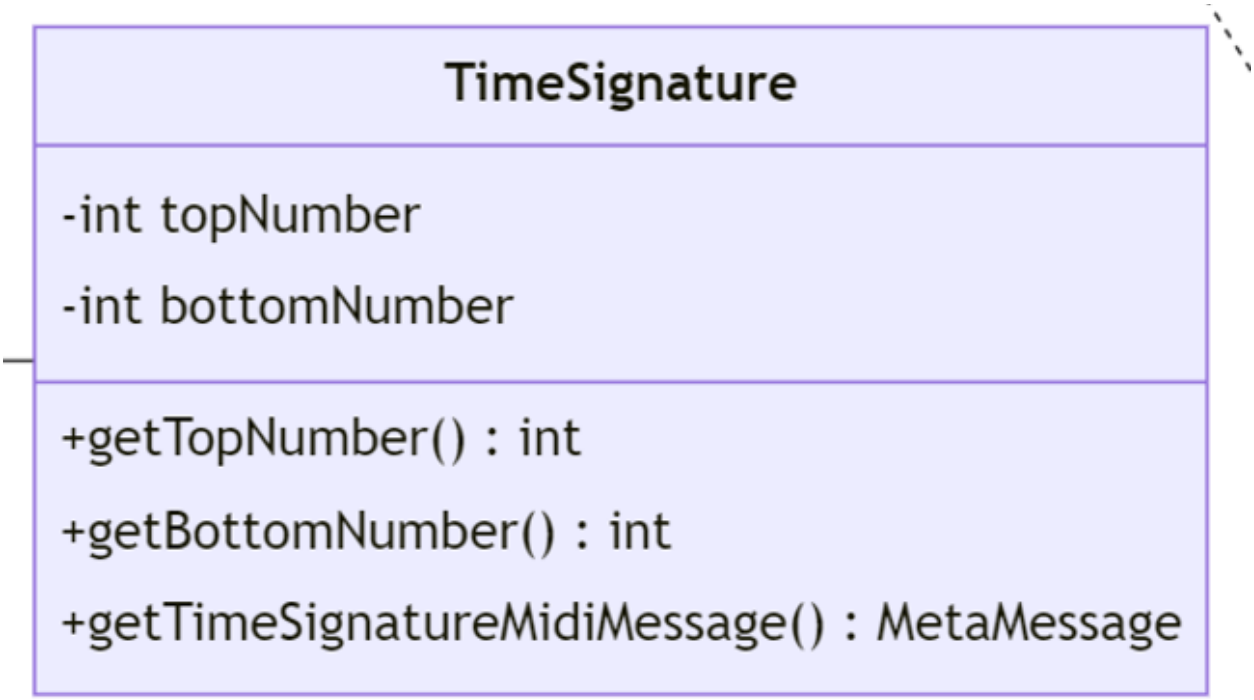
- RESOLUTION is a final int and is 64 (ticks per quarter note).
- generateSong() – Creates a random Chord (prevChord) as the initial previous Chord. Creates two "pseudo-Notes" using the protected Note constructor (prevMelodyNote and prevBassNote) based upon the midpoint of their value range. A for loop loops for numMeasures (the number of measures to be in the song), and does the following:
    - Makes a random chord.
    - If the random chord has the same root note as prevChord, another is randomly generated (so the chances of getting two chords in a row with the same root note become about 1 in 49).
    - Two Measures are created (nextMelodyMeasure and nextBassMeasure) based upon the information in Song's fields and the randomly generated chord.
    - nextMelodyMeasure and nextBassMeasure are added to their respective LinkedLists (melodyMeasures and bassMeasures).
    - prevChord is updated to be the randomly generated chord
    - If the Note returned from the getLastNonRestNote() method for nextMelodyMeasure is not null, then prevMelodyNote is updated with that Note.
    - If the Note returned from the getLastNonRestNote() method for nextBassMeasure is not null, then prevBassNote is updated with that Note.
- convertToSequence() – Returns a Sequence and throws InvalidMidiDataException. This method does the following:
    - Creates a Sequence (sequence) with a division type of Sequence.PPQ (pulses (ticks) per quarter note) and a resolution of RESOLUTION (64).
    - Creates a Track (track) for the sequence.
    - Creates a MetaMessage for the tempo of the sequence (bpmMessage) and uses it to create a MidiEvent which gets added to the Track (track).
    - Creates a MidiEvent for the time signature of the sequence using the MetaMessage obtained from the getTimeSignatureMidiMessage() method from the TimeSignature class and adds it to the Track (track).
    - If hasMelodyPart is true, then a for loop is used to iterate through the Measures in melodyMeasures, and a for loop is used to iterate through each Note in those Measures. Within the innermost for loop, two ShortMessage are created (startNote and stopNote) using the information from the Note (if it is not a rest) and used to make MidiEvents which are then added to the Track (track). A long (delay) is used to keep track of ticks as the loops iterate, and gets incremented as necessary.
    - If hasBassPart is true, then a for loop is used to iterate through the Measures in bassMeasures, and a for loop is used to iterate through each Note in those Measures. Within the innermost for loop, two ShortMessage are created (startNote and stopNote) using the information from the Note (if it is not a rest) and used to make MidiEvents which are then added to the Track (track). A long (delay) is used to keep track of ticks as the loops iterate, and gets incremented as necessary.
    - The completed Sequence (sequence) is returned.

- setRandomSeed(long randomSeed) – sets the seed for rand.
- Random rand – An instance of the Random class which gets passed to all the methods which need to generate random numbers to ensure consistent results when seeds are reused. Gets initialized in the class constructor, and the seed is set through a separate method.
- Class Constructor:

```
public Song(double BPM, int numMeasures, ArrayList<ArrayList<ArrayList<Double>>> ascendingNoteWeights,
        ArrayList<ArrayList<ArrayList<Double>>> descendingNoteWeights, ArrayList<Double> rhythmWeights,
        int melodyRangeMaxValue, int melodyRangeMinValue, int bassRangeMaxValue, int bassRangeMinValue,
        ScaleType scale, Notes scaleRootNote, TimeSignature timeSig, boolean hasMelodyPart, boolean hasBassPart,
        double restBias, double chordBias, double harmonicMelodicBias) {
```

  - The necessary fields are initialized.
  - There is no GUI formatting to be done.

class TimeSignature

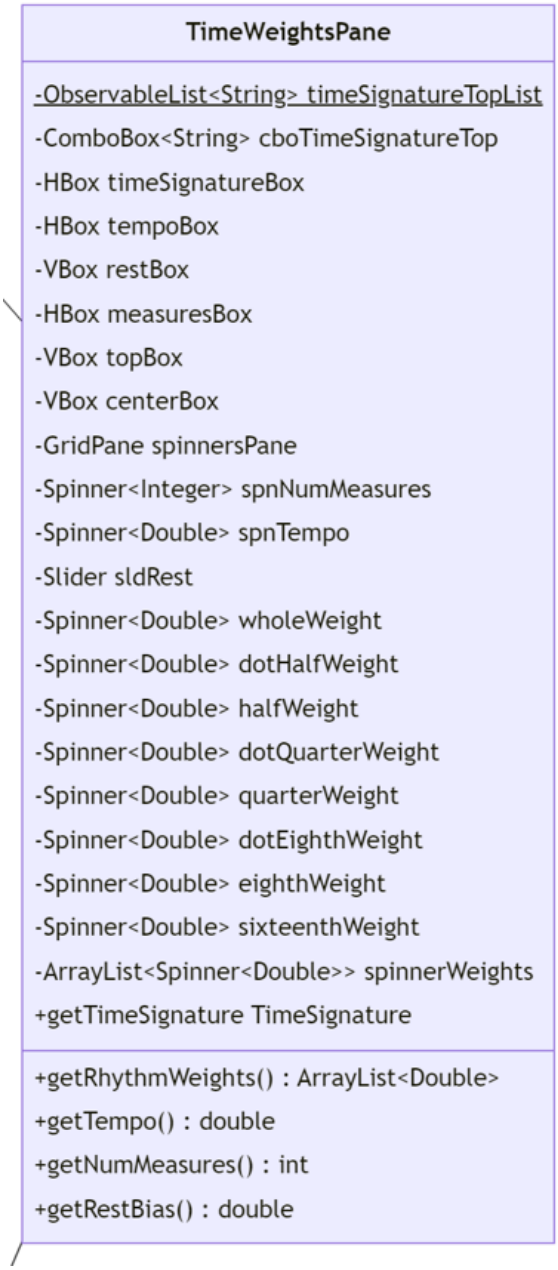


- getTopNumber() – Returns an int (topNumber) representing the numerator of a time signature.
- getBottomNumber() – Returns an int (bottomNumber) representing the denominator of a time signature.
- getTimeSignatureMetaMessage() – Returns a MetaMessage holding midi information for the time signature and throws InvalidMidiDataException. Uses a switch statement for topNumber to determine the byte value for the numerator in the MetaMessage.

- Class Constructor:

```
public TimeSignature(int topNumber, int bottomNumber) {
    this.topNumber = topNumber;
    this.bottomNumber = bottomNumber;
}
```

class TimeWeightsPane



- This class extends the BorderPane class from JavaFX.
- The ObservableList<String> (initNoteList) for the ComboBox<String> (cboTimeSignatureTop) is as follows:

```
private static ObservableList<String> timeSignatureTopList = FXCollections.observableArrayList("2", "3", "4", "5");
```

- getRhythmWeights() – Returns an ArrayList<Double> holding the double values from the Spinners for each respective note length type (the values are in order of whole note, dotted half note, half note, …, sixteenth note).
- getTempo() – Returns a double (the value from spnTempo and is 120.0 by default).
- getNumMeasures() – Returns an int (the value from spnNumMeasures and is 64 by default).
- getRestBias() – Returns a double (the value from sldRest and is 2.5 by default).
- getTimeSignature() – Returns a TimeSignature (created based upon the current value in cboTimeSignatureTop).
- Class Constructor:

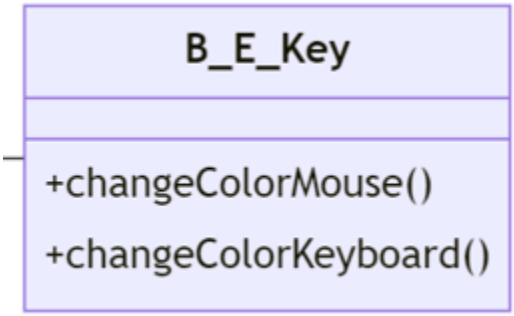
```
public TimeWeightsPane() {
    super();
```

  - The necessary fields are initialized.
  - The default weights for the Spinners corresponding with the different note length types can be seen here (the third value the Spinner constructors):

```
wholeWeight = new Spinner<Double>(0.0, 100.0, 1.0, 1.0);
wholeWeight.setEditable(true);
dotHalfWeight = new Spinner<Double>(0.0, 100.0, 2.0, 1.0);
dotHalfWeight.setEditable(true);
halfWeight = new Spinner<Double>(0.0, 100.0, 5.0, 1.0);
halfWeight.setEditable(true);
dotQuarterWeight = new Spinner<Double>(0.0, 100.0, 10.0, 1.0);
dotQuarterWeight.setEditable(true);
quarterWeight = new Spinner<Double>(0.0, 100.0, 30.0, 1.0);
quarterWeight.setEditable(true);
dotEighthWeight = new Spinner<Double>(0.0, 100.0, 5.0, 1.0);
dotEighthWeight.setEditable(true);
eighthWeight = new Spinner<Double>(0.0, 100.0, 32.0, 1.0);
eighthWeight.setEditable(true);
sixteenthWeight = new Spinner<Double>(0.0, 100.0, 15.0, 1.0);
sixteenthWeight.setEditable(true);
```

  - The Spinners for the different note length types are added to ArrayList<Spinner<Double>> spinnerWeights. The other spinners are not added to spinnerWeights.
  - The pane is formatted to look nice.

class B_E_Key



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).
- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.
    - For not mapped keys:

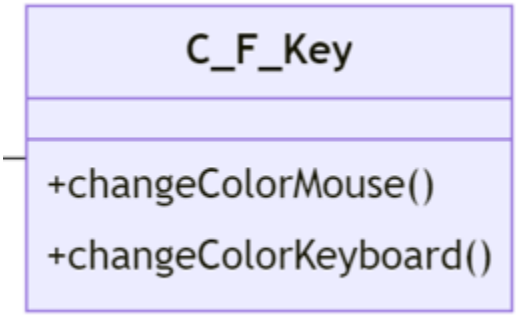
```
public B_E_Key(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

    - For mapped keys:

```
public B_E_Key(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight, KeyCode keyboardKey) {
    super(receiver, note, octave, keyboardKey);
```

        - The points for the polygon are specified here and the stroke and fill are set here. The end result looks like a B key or an E key on a piano.

class C_F_Key



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).
- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.
    - For not mapped keys:

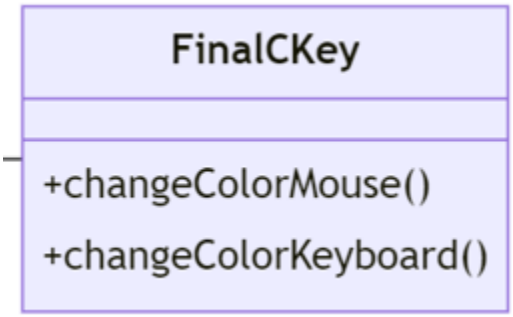
```
public C_F_Key(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

    - For mapped keys:

```
public C_F_Key(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight, KeyCode keyboardKey) {
    super(receiver, note, octave, keyboardKey);
```

        - The points for the polygon are specified here and the stroke and fill are set here. The end result looks like a C key or an F key on a piano.

class FinalCKey



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).
- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.
    - For not mapped keys:

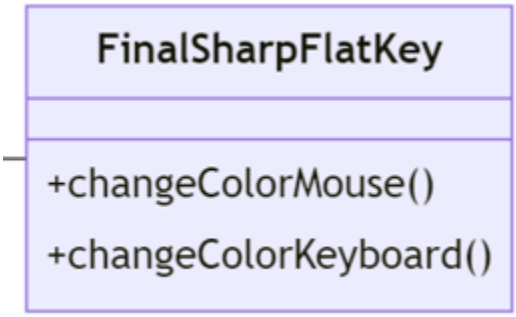
```
public FinalCKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

    - For mapped keys:

```
public FinalCKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight, KeyCode keyboardKey) {
    super(receiver, note, octave, keyboardKey);
```

        - The points for the polygon are specified here and the stroke and fill are set here. The end result looks the C8 key on a piano.

class FinalSharpFlatKey



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from darker dark gray to even darker dark gray, or vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).
- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from darker dark gray to even darker dark gray, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.
    - For not mapped keys:

```
public FinalSharpFlatKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

    - For mapped keys:

```
public FinalSharpFlatKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight, KeyCode keyboardKey) {
    super(receiver, note, octave, keyboardKey);
```

        - The points for the polygon are specified here and the stroke and fill are set here. The end result looks the left half of a black key on a piano.

class KeyboardPane

- This class extends the Pane class from JavaFX.
- The Map fields for this class are as follows:

```
private static Map<Integer, String> sharpKeyLabelMap = Map.ofEntries(
        entry(k: 0, v: "Q"),
        entry(k: 1, v: "W"),
        entry(k: 2, v: "E"),
        entry(k: 3, v: "R"),
        entry(k: 4, v: "T"),
        entry(k: 5, v: "Y"),
        entry(k: 6, v: "U"),
        entry(k: 7, v: " I"),
        entry(k: 8, v: "O"),
        entry(k: 9, v: "P"),
        entry(k: 10, v: "["));
```

```
private static Map<Integer, String> regularKeyLabelMap = Map.ofEntries(
        entry(k: 0, v: "A"),
        entry(k: 1, v: "S"),
        entry(k: 2, v: "D"),
        entry(k: 3, v: "F"),
        entry(k: 4, v: "G"),
        entry(k: 5, v: "H"),
        entry(k: 6, v: "J"),
        entry(k: 7, v: "K"),
        entry(k: 8, v: "L"),
        entry(k: 9, v: " ;"));
```

```
private static Map<Integer, KeyCode> regularKeyMap = Map.ofEntries(
        entry(k: 0, KeyCode.A),
        entry(k: 1, KeyCode.S),
        entry(k: 2, KeyCode.D),
        entry(k: 3, KeyCode.F),
        entry(k: 4, KeyCode.G),
        entry(k: 5, KeyCode.H),
        entry(k: 6, KeyCode.J),
        entry(k: 7, KeyCode.K),
        entry(k: 8, KeyCode.L),
        entry(k: 9, KeyCode.SEMICOLON));
```

```
private static Map<Integer, KeyCode> sharpKeyMap = Map.ofEntries(
        entry(k: 0, KeyCode.Q),
        entry(k: 1, KeyCode.W),
        entry(k: 2, KeyCode.E),
        entry(k: 3, KeyCode.R),
        entry(k: 4, KeyCode.T),
        entry(k: 5, KeyCode.Y),
        entry(k: 6, KeyCode.U),
        entry(k: 7, KeyCode.I),
        entry(k: 8, KeyCode.O),
        entry(k: 9, KeyCode.P),
        entry(k: 10, KeyCode.OPEN_BRACKET));
```

- setDisplayKeyLabels(boolean display) and setDisplayMappingLabels(boolean display) set their associated fields with the appropriate value, then call the paintPiano() method.
- stepUpOctaveDisplayed(), stepDownOctaveDisplayed(), stepUpNoteDisplayed(), stepDownNoteDisplayed() – These methods raise/lower the values for bottomNoteDisplayed and bottomNoteDisplayedOctave as necessary. The piano does not get repainted.
- stepUpOctaveMapped(), stepDownOctaveMapped(), stepUpNoteMapped(), stepDownNoteMapped() – These methods raise/lower the values for bottomNoteMapped and bottomNoteMappedOctave as necessary. The piano does not get repainted.
- compareNotes(Notes note1, int octave1, Notes note2, int octave2) – Returns an int value (either -1, 0, or 1). 0 is returned if the two notes have equal note names and octaves. -1 is returned if note1 is lower than note2, 1 is returned if note 1 is higher than note2.
- paintPiano() – This is the most important method of this class! It does the following:
    - Stops all the notes by calling the stopAllNotes() method.
    - Clears all the current children of the KeyboardPane and clears all the elements in mappedKeys, normalKeys, mappingLabels, and keyLabels.
    - Determines the height and width of each note depending upon the current size of the pane and numRegularKeys (the number of white piano keys to be displayed).
    - Uses a for loop and switch statements to fill normalKeys with instances of the different subclasses of PianoKey using the constructor for not mapped notes.
        - Has a nested for loop which triggers when the note to be added next matches with bottomNoteMapped and bottomNoteMappedOctave (determined by using the compareNotes() method).
        - This nested loop fills mappedKeys with instances of the different subclasses of PianoKey using the constructor for mapped notes while generating their corresponding mapping Label alongside them and adding that to mappingLabels.
        - Once the nested loop finished, the outer loop continues to generate instances of PianoKeys until the desired number of keys is reached.

    - Once the loops finish, the contents of mappedKeys and normalKeys are added to the pane. If displayMappingLabels is true, each Label in mappingLabels is set to be mouse transparent and added to the pane.
    - If displayKeyLabels is true, a for loop with a switch statement adds mouse transparent labels to keyLabels for the note name of each white key. Then the contents of keyLabels gets added to the pane.
- subtractNoteSteps(Notes note, int octave, int stepsDown) – Returns a Pair<Notes, Integer> which represents the value of the new note after the specified number of steps have been traveled downward. Accomplishes this using a for loop and switch statement.
- addNoteSteps(Notes note, int octave, int stepsDown) – Returns a Pair<Notes, Integer> which represents the value of the new note after the specified number of steps have been traveled upward. Accomplishes this using a for loop and switch statement.
- stopAllNotes() – Calls the stopNote() method for every PianoKey in mappedKeys.
- stepDownNoteMapping(), stepDownOctaveMapping(), stepUpNoteMapping() stepUpOctaveMapping() – These methods step up/down the values for bottomNoteMapped and bottomNoteOctaveMapped using the private methods which are similarly named (stepUpOctaveMapped(), stepDownOctaveMapped(), stepUpNoteMapped(), stepDownNoteMapped()). However, the value is adjusted to ensure that the range of mapped notes do not go outside the range of the piano (A0 to C8) and bottomNoteDisplayed and bottomNoteDisplayedOctave are adjusted as well if needed so that the mapped notes are always displayed. The piano is then repainted using the paintPiano() method.
- stepDownNoteDisplay(), stepDownOctaveDisplay(), stepUpNoteDisplay(), stepUpOctaveDisplay() – These methods step up/down the values for bottomNoteDisplayed and bottomNoteOctaveDisplayed using the private methods which are similarly named (stepUpOctaveDisplayed(), stepDownOctaveDisplayed(), stepUpNoteDisplayed(), stepDownNoteDisplayed(). However, the value is adjusted to ensure that the range of displayed notes do not go outside the range of the piano (A0 to C8) and bottomNoteMapped and bottomNoteMappedOctave are adjusted as well if needed so that the mapped notes are always displayed. The piano is then repainted using the paintPiano() method.
- addNote() – Adds a note to be displayed (maximum is 52 because a piano has 52 white keys). The notes are added to the top of the display first, but if the top note displayed reaches C8 (the top note on a piano), the following notes will be added to the bottom of the display. The piano is then repainted using the paintPiano() method.
- removeNote() – Removes a note from the notes display (minimum is 10 because there are 10 mapped white keys and all mapped keys must be displayed). The notes are removed from the top note displayed and the key mapping is adjusted downward if necessary to ensure that all mapped keys are displayed.
- setWidth(double width) – Overrides the setWidth(double width) method from the parent class. Calls super.setWidth(width), and then calls the paintPiano() method.
- setHeight(double height) – Overrides the setWidth(double height) method from the parent class. Calls super.setHeight (height), and then calls the paintPiano() method.

- getMappedKeys() – Returns an ArrayList<PianoKey> (mappedKeys).
- Class Constructors:
    - The following constructor has the bottom note displayed and mapped as A3:

```
public KeyboardPane(Receiver receiver) {
    super();
    this.receiver = receiver;
    paintPiano();

}
```

    - The following constructor allows for the bottom note displayed and mapped to be input as an argument in the constructor (if the input note is not in the acceptable range (A0 to C8), it is adjusted to fit within the range of a piano. The paintPiano() method is called at the end of the constructor.):

```
public KeyboardPane(Notes note, int octave, Receiver receiver) {
    super();
    this.receiver = receiver;
```

class PianoKey

- This class is an abstract class and it extends the Polygon class from JavaFX.
- getVolumeProperty() – Returns an IntegerProperty (volume).
- startNote() – Sends a ShortMessage to the Receiver (receiver) to start the note with the value associated with this key at the volume specified by the IntegerProperty (volume).
- stopNote() – Sends a ShortMessage to the Receiver (receiver) to stop the note with the value associated with this key at the volume specified by the IntegerProperty (volume).
- changeColorMouse() – An abstract method to be overridden by subclasses.
- changeColorKeyboard() – An abstract method to be overridden by subclasses.
- setMouseHandling() – Sets the actions for different mouse inputs (clicking/dragging) so that the piano keys function as intended (stop when the mouse button is released or dragged to a different key, start when the mouse button is pressed or dragged over that key). Makes use of the changeColorMouse() and setIsMousePressed() methods.
- getKeyboardKey() – Returns the KeyCode (keyboardKey) associated with this piano key (null if it is not mapped).
- getIsMousePressed() – Returns a boolean (isMousePressed).
- setIsMousePressed(boolean pressed) – Sets the value for the boolean (isMousePressed).
- getIsKeyboardPressed() – Returns a boolean (isKeyboardPressed).
- setIsKeyboardPressed(boolean pressed) – Sets the value for the boolean (isKeyboardPressed).
- getNote() – Returns a Notes (note, which is the note name associated with this key).
- getOctave() – Returns an int (octave, which is the octave of the note associated with this key).
- Class Constructor:

```
public PianoKey(Receiver receiver, Notes note, int octave, KeyCode keyboardKey) {
    super();
    this.keyboardKey = keyboardKey;
    this.receiver = receiver;
    this.note = note;
    this.octave = octave;
    this.setMouseHandling();
}
```

class PianoPane

- This class extends the BorderPane class from JavaFX.
- The ObservableList<String> (instruments) for this class is as follows:

```
private static ObservableList<String> instruments = FXCollections.observableArrayList("Piano", "Harpsichord",
        "Music Box", "Church Organ", "Guitar", "Electric Bass", "Violin", "Trumpet", "Square", "Sawtooth");
```

- The Map<String, Integer> (instrumentMap) for this class is as follows (it maps instrument names to their midi code):

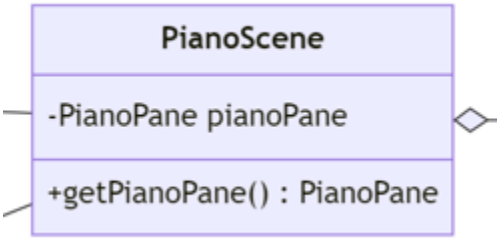
```
private static Map<String, Integer> instrumentMap = Map.ofEntries(
        entry(k: "Piano", v: 0),
        entry(k: "Harpsichord", v: 6),
        entry(k: "Music Box", v: 10),
        entry(k: "Church Organ", v: 19),
        entry(k: "Guitar", v: 25),
        entry(k: "Electric Bass", v: 33),
        entry(k: "Violin", v: 41),
        entry(k: "Trumpet", v: 56),
        entry(k: "Square", v: 80),
        entry(k: "Sawtooth", v: 81));
```

- getVolumeSlider() – Returns a Slider (sldVolume).
- getKeyboardPane() – Returns a KeyboardPane (piano).
- getExitButton() – Returns a Button (btnExit).
- getButtonsPanelTop() – Returns a VBox (buttonsPanelTop).
- getButtonsPanelBottom() – Returns a VBox (buttonsPanelBottom).
- getReceiver() – Returns a Receiver (receiver).
- Class Constructor:

```
public PianoPane(Receiver receiver) {
    super();
    this.receiver = receiver;
```

    - The necessary fields are initialized.
    - buttonsPanelTop holds the buttons btnAddNote, btnRemoveNote, btnDownOctaveMapped, btnDownStepMapped, btnUpStepMapped, btnUpOctaveMapped, btnDownOctaveDisplay, btnDownStepDisplay, btnUpStepDisplay, btnUpOctaveDisplay.
    - The buttons contained in buttonsPanelTop each have their actions set to call the methods for the KeyboardPane (piano) associated with that particular button (For example, btnAddNote does piano.addNote().)
    - buttonsPanelBottom holds cboInstruments and btnExit.
    - The pane is formatted to look nice.

class PianoScene

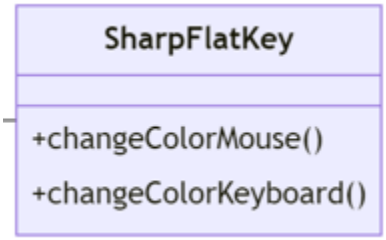


- This class extends the Scene class from JavaFX.
- getPianoPane() – Returns a PianoPane (pianoPane).
- Class Constructor:

```
public PianoScene(double width, double height, Receiver receiver) {
  super(new PianoPane(receiver), width, height);
```

  - Sets actions for scene to do OnKeyPressed and OnKeyReleased.
    - Iterates through the PianoKeys in mappedKeys from the KeyboardPane held in the PianoPane.
    - If the PianoKey's KeyCode (from the getKeyboardKey() method) matches the KeyCode from the event, then the stopNote() or startNote() method is called for that PianoKey. Then the PianoKey's changeColorKeyboard() and setIsKeyboardPressed(pressed) are also called, with the value for pressed being true or false as necessary depending on the event.
    - OnKeyPressed also checks for the KeyCodes for Z, X, C, and V.
      - Z calls the KeyboardPane's stepDownOctaveMapping() method.
      - X calls the KeyboardPane's stepDownNoteMapping() method.
      - C calls the KeyboardPane's stepUpNoteMapping() method.
      - V calls the KeyboardPane's stepUpOctaveMapping() method.

class SharpFlatKey



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from darker dark gray to even darker dark gray, or

vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).

- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from darker dark gray to even darker dark gray, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.
    - For not mapped keys:

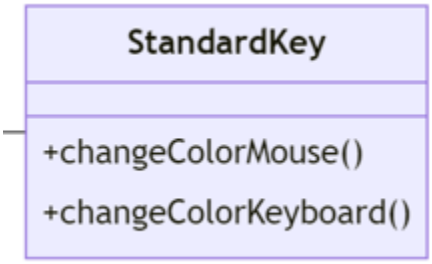
```
public SharpFlatKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

    - For mapped keys:

```
public SharpFlatKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight, KeyCode keyboardKey) {
    super(receiver, note, octave, keyboardKey);
```

        - The points for the polygon are specified here and the stroke and fill are set here. Additionally, it checks if it will appear as the first black key display (as in, right up against the edge of the window). The end result looks a black key on a piano (or the right half of a black key on a piano if it is against the edge of the window).

class StandardKey



- This class extends the abstract PianoKey class.
- changeColorMouse() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsMousePressed() method from the parent class).
- changeColorKeyboard() – This method overrides the abstract method from the PianoKey class. It changes the key's fill color from white to darker white, or vice versa, depending upon if the mouse is pressed (determined using the getIsKeyboardPressed() method from the parent class).
- Class Constructors – This class has two constructors, one used for mapped keys and the other used for when the key is not a mapped key.

- For not mapped keys:

```
public C_F_Key(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

- For mapped keys:

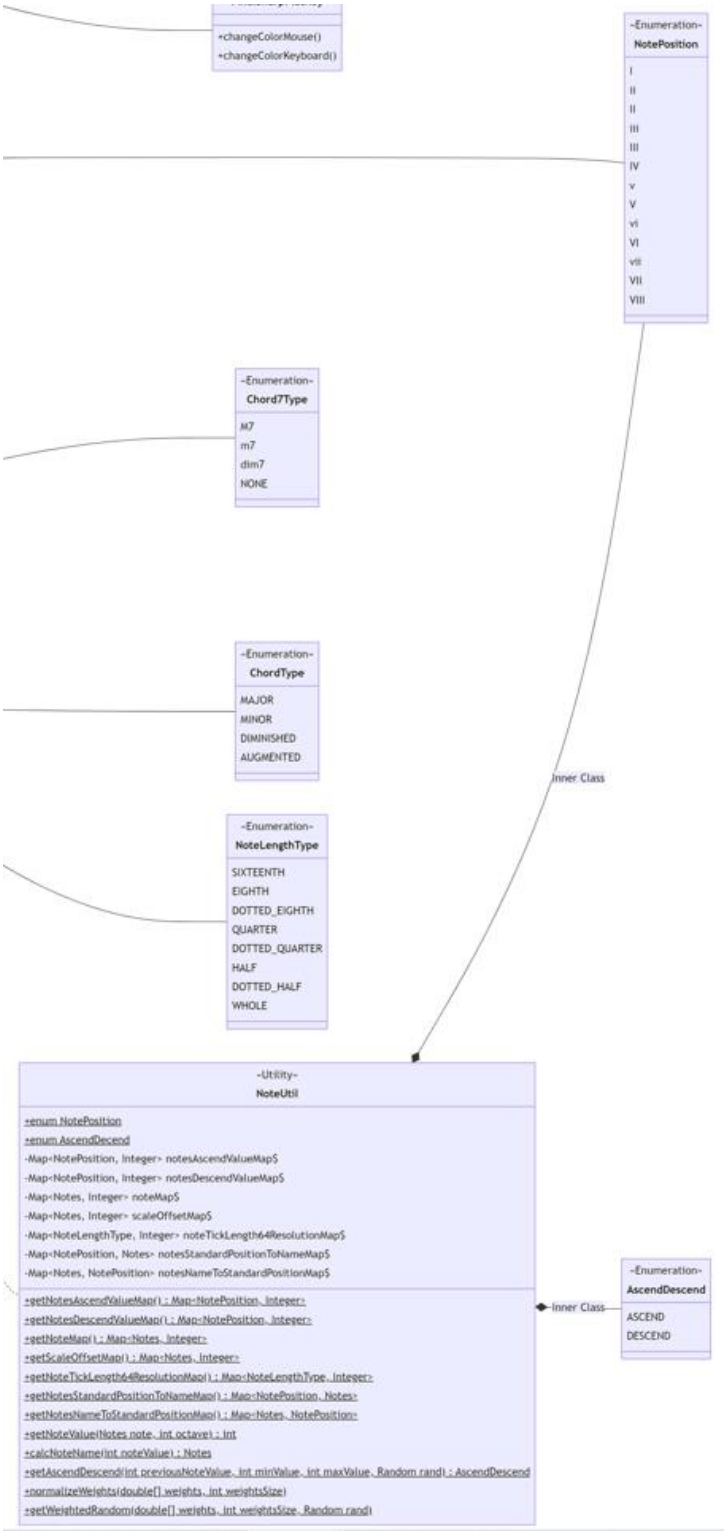
```
public StandardKey(Double leftAnchor, Double topAnchor, Receiver receiver, Notes note, int octave,
        double noteWidth, double noteHeight) {
    this(leftAnchor, topAnchor, receiver, note, octave, noteWidth, noteHeight, keyboardKey: null);
}
```

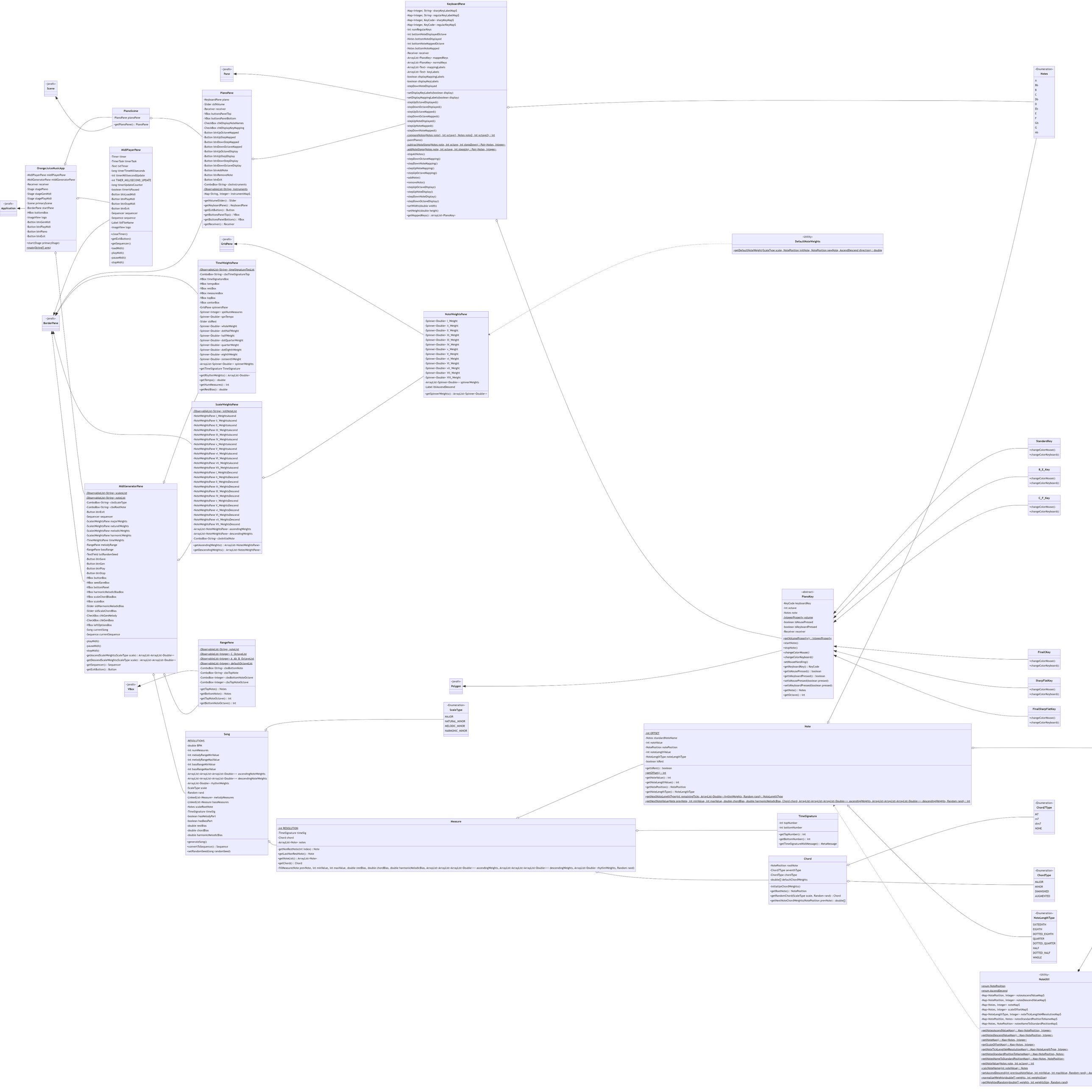
  - The points for the polygon are specified here and the stroke and fill are set here. The end result looks like a D key, a G key or an A key on a piano.

Orange Juice Music App – Implementation Summary

The main class is OrangeJuiceMusicApp, which will open a stage holding MidiPlayerPane, a stage holding MidiGeneratorPane, and a stage holding PianoScene. MidiPlayerPane plays midi files and has a simple GUI. MidiGeneratorPane creates, plays, and saves midi files. MidiGeneratorPane holds TimeWeightsPane, ScaleWeightsPane, and RangePane. TimeWeightsPane holds the GUI elements related to time. ScaleWeightsPane holds NoteWeightsPane. NoteWeightsPane makes use of DefaultNoteWeights and holds GUI elements related to note pitch. RangePane holds GUI elements related to the note pitch bounds. MidiGeneratorPane creates Song. Song contains Measure, and Measure contains Note. Song contains the data to generate many instances of Measure, and Measure contains the data to generate many instances of Note. Song contains ScaleType (and technically TimeSignature), Measure contains Chord and TimeSignature, and Note contains the information necessary to be added as a midi event to a sequence. Note makes use of NoteUtil. NoteUtil contains static methods to allow for the purpose of allowing clarity in other classes or because the methods did not seem to fit well in any other classes because multiple fairly unrelated classes make use of that method. PianoScene displays a virtual piano. PianoScene holds PianoPane and handles keyboard input. PianoPane contains KeyboardPane and GUI elements to modify KeyboardPane. KeyboardPane holds subclasses of PianoKey (an abstract class), namely: StandardKey, B_E_Key, C_F_Key, FinalCKey, SharpFlatKey, FinalSharpFlatKey. PianoKey handles mouse input on piano keys.

Orange Juice Music App – Final UML Diagram




KeyboardPane
-Map<Integer, String> sharpKeyLabelMap$
-Map<Integer, String> regularKeyLabelMap$
-Map<Integer, KeyCode> sharpKeyMap$
-Map<Integer, KeyCode> regularKeyMap$
-int numRegularKeys
-int bottomNoteDisplayedOctave
-Notes bottomNoteDisplayed
-Int bottomNoteMappedOctave
-Notes bottomNoteMapped
-Receiver receiver
-ArrayList<PianoKey> mappedKeys
-ArrayList<PianoKey> normalKeys
-ArrayList<Text> mappingLabels
-ArrayList<Text> keyLabels
-boolean displayMappingLabels
-boolean displayKeyLabels
-stepDownNoteDisplayed
+setDisplayKeyLabels(boolean display)
+setDisplayMappingLabels(boolean display)
-stepUpOctaveDisplayed()
-stepDownOctaveDisplayed()
-stepUpOctaveMapped()
-stepDownOctaveMapped()
-stepUpNoteDisplayed()
-stepUpNoteMapped()
-stepDownNoteMapped()
-compareNotes(Notes note1, int octave1, Notes note2, int octave2) : Int
-paintPiano()
-subtractNoteSteps(Notes note, int octave, int stepsDown) : Pair<Notes, Integer>
-addNoteSteps(Notes note, int octave, int stepsUp) : Pair<Notes, Integer>
-stopAllNotes()
+stepDownOctaveMapping()
+stepDownNoteMapping()
+stepUpNoteMapping()
+stepUpOctaveMapping()
+addNote()
+removeNote()
+stepUpOctaveDisplay()
+stepUpNoteDisplay()
+stepDownNoteDisplay()
+stepDownOctaveDisplay()
+setWidth(double width)
+setHeight(double height)
+getMappedKeys() : ArrayList<PianoKey>
~javafx~
Pane
~javafx~
Scene
PianoScene
-PianoPane pianoPane
+getPianoPane() : PianoPane
PianoPane
-KeyboardPane piano
-Slider sldVolume
-Receiver receiver
-VBox buttonsPanelTop
-VBox buttonsPanelButtom
-CheckBox chkDisplayNoteNames
-CheckBox chkDisplayKeyMapping
-Button btnUpOctaveMapped
-Button btnUpStepMapped
-Button btnDownStepMapped
-Button btnDownOctaveMapped
-Button btnUpOctaveDisplay
-Button btnUpStepDisplay
-Button btnDownStepDisplay
-Button btnDownOctaveDisplay
-Button btnAddNote
-Button btnRemoveNote
-Button btnExit
-ComboBox<String> cboInstruments
-ObservableList<String> instruments
-Map<String, Integer> instrumentMap$
+getVolumeSlider() : Slider
+getKeyboardPane() : KeyboardPane
+getExitButton() : Button
+getButtonsPanelTop() : VBox
+getButtonsPanelBottom() : VBox
+getReceiver() : Receiver
~Enumeration~
Notes
A
Bb
B
C
Db
D
Eb
E
F
Gb
G
Ab
MidiPlayerPane
-Timer timer
-TimerTask timerTask
-Text txtTimer
-long timerTimeMilliseconds
-int timerMillisecondUpdate
-int TIMER_MILLISECOND_UPDATE
-long timerUpdateCounter
-boolean timerIsPaused
-Button btnLoadMidi
-Button btnPlayMidi
-Button btnStopMidi
-Button btnExit
-Sequencer sequencer
-Sequence sequence
-Label lblFileName
-ImageView logo
+closeTimer()
+getExitButton()
+getSequencer()
-loadMidi()
-playMidi()
-pauseMidi()
-stopMidi()
OrangeJuiceMusicApp
-MidiPlayerPane midiPlayerPane
-MidiGeneratorPane midiGeneratorPane
-Receiver receiver
-Stage stagePiano
-Stage stageGenMidi
-Stage stagePlayMidi
-Scene primaryScene
-BorderPane startPane
-HBox buttonsBox
-ImageView logo
-Button btnGenMidi
-Button btnPlayMidi
-Button btnPiano
-Button btnExit
+start(Stage primaryStage)
+main(String[] args)
~javafx~
Application
~javafx~
GridPane
~javafx~
BorderPane
~Utility~
DefaultNoteWeights
+getDefaultNoteWeight(ScaleType scale, NotePosition initNote, NotePosition newNote, AscendDescend direction) : double
TimeWeightsPane
-ObservableList<String> timeSignatureTopList
-ComboBox<String> cboTimeSignatureTop
-HBox timeSignatureBox
-HBox tempoBox
-VBox restBox
-HBox measuresBox
-VBox topBox
-VBox centerBox
-GridPane spinnersPane
-Spinner<Integer> spnNumMeasures
-Spinner<Double> spnTempo
-Slider sldRest
-Spinner<Double> wholeWeight
-Spinner<Double> dotHalfWeight
-Spinner<Double> halfWeight
-Spinner<Double> dotQuarterWeight
-Spinner<Double> quarterWeight
-Spinner<Double> dotEighthWeight
-Spinner<Double> eighthWeight
-Spinner<Double> sixteenthWeight
-ArrayList<Spinner<Double>> spinnerWeights
+getTimeSignature TimeSignature
+getRhythmWeights() : ArrayList<Double>
+getTempo() : double
+getNumMeasures() : Int
+getRestBias() : double
NoteWeightsPane
-Spinner<Double> I_Weight
-Spinner<Double> II_Weight
-Spinner<Double> II_Weight
-Spinner<Double> III_Weight
-Spinner<Double> III_Weight
-Spinner<Double> IV_Weight
-Spinner<Double> v_Weight
-Spinner<Double> V_Weight
-Spinner<Double> vi_Weight
-Spinner<Double> VI_Weight
-Spinner<Double> vii_Weight
-Spinner<Double> VII_Weight
-Spinner<Double> VIII_Weight
-ArrayList<Spinner<Double>> spinnerWeights
-Label lblAscendDescend
+getSpinnerWeights() : ArrayList<Spinner<Double>>
ScaleWeightsPane
-ObservableList<String> initNoteList
-NoteWeightsPane I_WeightsAscend
-NoteWeightsPane II_WeightsAscend
-NoteWeightsPane II_WeightsAscend
-NoteWeightsPane III_WeightsAscend
-NoteWeightsPane III_WeightsAscend
-NoteWeightsPane IV_WeightsAscend
-NoteWeightsPane v_WeightsAscend
-NoteWeightsPane V_WeightsAscend
-NoteWeightsPane vi_WeightsAscend
-NoteWeightsPane VI_WeightsAscend
-NoteWeightsPane vii_WeightsAscend
-NoteWeightsPane VII_WeightsAscend
-NoteWeightsPane I_WeightsDescend
-NoteWeightsPane II_WeightsDescend
-NoteWeightsPane II_WeightsDescend
-NoteWeightsPane III_WeightsDescend
-NoteWeightsPane III_WeightsDescend
-NoteWeightsPane IV_WeightsDescend
-NoteWeightsPane v_WeightsDescend
-NoteWeightsPane V_WeightsDescend
-NoteWeightsPane vi_WeightsDescend
-NoteWeightsPane VI_WeightsDescend
-NoteWeightsPane vii_WeightsDescend
-NoteWeightsPane VII_WeightsDescend
-ArrayList<NoteWeightsPane> ascendingWeights
-ArrayList<NoteWeightsPane> descendingWeights
-ComboBox<String> cboInitialNote
+getAscendingWeights() : ArrayList<NotesWeightsPane>
+getDescendingWeights() : ArrayList<NotesWeightPane>
MidiGeneratorPane
-ObservableList<String> scalesList
-ObservableList<String> noteList
-ComboBox<String> cboScaleType
-ComboBox<String> cboRootNote
-Button btnExit
-Sequencer sequencer
-ScalesWeightsPane majorWeights
-ScalesWeightsPane naturalWeights
-ScalesWeightsPane melodicWeights
-ScalesWeightsPane harmonicWeights
-TimeWeightsPane timeWeights
-RangePane melodyRange
-RangePane bassRange
-TextField txtRandomSeed
-Button btnSave
-Button btnGen
-Button btnPlay
-Button btnStop
-HBox buttonBox
-HBox seedSaveBox
-VBox bottomPanel
-VBox harmonicMelodicBiasBox
-VBox scaleChordBiasBox
-VBox scaleBox
-Slider sldHarmonicMelodicBias
-Slider sldScaleChordBias
-CheckBox chkGenMelody
-CheckBox chkGenBass
-VBox leftOptionsBox
-Song currentSong
-Sequence currentSequence
-playMidi()
-pauseMidi()
-stopMidi()
-getAscendScaleWeights(ScaleType scale) : ArrayList<ArrayList<Double>>
-getDescendScaleWeights(ScaleType scale) : ArrayList<ArrayList<Double>>
-getSequencer() : Sequencer
-getExitButton() : Button
RangePane
-ObservableList<String> noteList
-ObservableList<Integer> C_OctaveList
-ObservableList<Integer> A_Ab_B_OctaveList
-ObservableList<Integer> defaultOctaveList
-ComboBox<String> cboBottomNote
-ComboBox<String> cboTopNote
-ComboBox<Integer> cboBottomNoteOctave
-ComboBox<Integer> cboTopNoteOctave
+getTopNote() : Notes
+getBottomNote() : Notes
+getTopNoteOctave() : int
+getBottomNoteOctave() : int
~javafx~
VBox
~javafx~
Polygon
~Enumeration~
ScaleType
MAJOR
NATURAL_MINOR
MELODIC_MINOR
HARMONIC_MINOR
Song
-RESOLUTION$
-double BPM
-int numMeasures
-int melodyRangeMinValue
-int melodyRangeMaxValue
-int bassRangeMinValue
-int bassRangeMaxValue
-ArrayList<ArrayList<ArrayList<Double>>> ascendingNoteWeights
-ArrayList<ArrayList<ArrayList<Double>>> descendingNoteWeights
-ArrayList<Double> rhythmWeights
-ScaleType scale
-Random rand
-LinkedList<Measure> melodyMeasures
-LinkedList<Measure bassMeasures
-Notes scaleRootNote
-TimeSignature timeSig
-boolean hasMelodyPart
-boolean hasBassPart
-double restBias
-double chordBias
-double harmonicMelodicBias
+generateSong()
+convertToSequence() : Sequence
+setRandomSeed(long randomSeed)
Measure
-int RESOLUTION
-TimeSignature timeSig
-Chord chord
-ArrayList<Note> notes
-getNonRestNote(int Index) : Note
+getLastNonRestNote() : Note
+getNoteList() : ArrayList<Note>
+getChord() : Chord
-fillMeasure(Note prevNote, int minValue, int maxValue, double restBias, double chordBias, double harmonicMelodicBias, ArrayList<ArrayList<ArrayList<Double<<< ascendingWeights, ArrayList<ArrayList<ArrayList<Double>>> descendingWeights, ArrayList<Double> rhythmWeights, Random rand)
Note
-int OFFSET
-Notes standardNoteName
-int noteValue
-NotePosition notePosition
-int noteLengthValue
-NoteLengthType noteLengthType
-boolean isRest
+getIsRest() : boolean
+getOffset() : int
+getNoteValue() : Int
+getNoteLengthValue() : int
+getNotePosition() : NotePosition
+getNoteLengthType() : NoteLengthType
+getNextNoteLength(Type(int remainingTicks, ArrayList<Double> rhythmWeights, Random rand) : NoteLengthType
+getNextNoteValue(Note prevNote, int minValue, int maxValue, double chordBias, double harmonicMelodicBias, Chord chord, ArrayList<ArrayList<ArrayList<Double<<< ascendingWeights, ArrayList<ArrayList<ArrayList<Double>>> descendingWeights, Random rand) : Int
TimeSignature
-Int topNumber
-int bottomNumber
+getTopNumber() : Int
+getBottomNumber() : Int
+getTimeSignatureMidiMessage() : MetaMessage
Chord
-NotePosition rootNote
-ChordType seventhType
-ChordType chordType
-double[] defaultChordWeights
-InitializeChordWeights()
+getRootNote() : NotePosition
+getRandomChord(ScaleType scale, Random rand) : Chord
+getNextNoteChordWeights(NotePosition prevNotes) : double[]
~abstract~
PianoKey
-KeyCode keyboardKey
-int octave
-Notes note
-IntegerProperty volume
-boolean isMousePressed
-boolean isKeyboardPressed
-Receiver receiver
+getVolumeProperty() : IntegerProperty
+startNote()
+stopNote()
+changeColorMouse()
+changeColorKeyboard()
-setMouseHandling()
+getKeyboardKey() : KeyCode
+getIsMousePressed() : boolean
+getIsKeyboardPressed() : boolean
+setIsMousePressed(boolean pressed)
+setIsKeyboardPressed(boolean pressed)
+getNote() : Notes
+getOctave() : int
StandardKey
+changeColorMouse()
+changeColorKeyboard()
B_E_Key
+changeColorMouse()
+changeColorKeyboard()
C_F_Key
+changeColorMouse()
+changeColorKeyboard()
FinalCKey
+changeColorMouse()
+changeColorKeyboard()
SharpFlatKey
+changeColorMouse()
+changeColorKeyboard()
FinalSharpFlatKey
+changeColorMouse()
+changeColorKeyboard()
~Enumeration~
Chord7Type
M7
m7
dim7
NONE
~Enumeration~
ChordType
MAJOR
MINOR
DIMINISHED
AUGMENTED
~Enumeration~
NoteLengthType
SIXTEENTH
EIGHTH
DOTTED_EIGHTH
QUARTER
DOTTED_QUARTER
HALF
DOTTED_HALF
WHOLE
~Utility~
NoteUtil
+enum NotePosition
+enum AscendDecend
-Map<NotePosition, Integer> notesAscendValueMap$
-Map<NotePosition, Integer> notesDescendValueMap$
-Map<Notes, Integer> noteMap$
-Map<Notes, Integer> scaleOffSetMap$
-Map<NoteLengthType, Integer> noteTickLength64ResolutionMap$
-Map<NotePosition, Notes> notesStandardPositionToNameMap$
-Map<Notes, NotePosition> notesNameToStandardPositionMap$
+getNotesAscendValueMap() : Map<NotePosition, Integer>
+getNotesDescendValueMap() : Map<NotePosition, Integer>
+getNoteMap() : Map<Notes, Integer>
+getScaleOffSetMap() : Map<Notes, Integer>
+getNoteTickLength64ResolutionMap() : Map<NoteLengthType, Integer>
+getNotesStandardPositionToNameMap() : Map<NotePosition, Notes>
+getNotesNameToStandardPositionMap() : Map<Notes, NotePosition>
+getNoteValue(Notes note, int octave) : int
+calcNoteName(int noteValue) : Notes
+getAscendDescend(int previousNoteValue, int minValue, int maxValue, Random rand) : AscendDescend
-normalizeWeights(double[] weights, int weightsSize)
+getWeightedRandom(double[] weights, int weightsSize, Random rand)