```
/*
==============================================================================
    CustomLookAndFeel.h
    Created: 6 Sep 2023 1:52:27am
    Author:  Ali

==============================================================================
*/

#pragma once

#include <JuceHeader.h>

class CustomLookAndFeel : public juce::LookAndFeel_V4
{
public:
    void drawButtonBackground (juce::Graphics& g, juce::Button& button,
                               const juce::Colour& backgroundColour, bool
                    isMouseOverButton, bool isButtonDown) override;

    void drawButtonText (juce::Graphics& g, juce::TextButton& button, bool
      isMouseOverButton, bool isButtonDown) override;

    void drawLinearSlider (juce::Graphics& g, int x, int y, int width, int
      height,
                           float sliderPos, float minSliderPos, float
                    maxSliderPos,
                           const juce::Slider::SliderStyle style,
                    juce::Slider& slider) override;

    void drawWaveform(juce::Graphics& g, const juce::Rectangle<int>& area,
      const float* data, int dataSize);

private:
    juce::Font getFontFromHeight(int height, const juce::String& text);
};
```

```cpp
/*
==============================================================================

    DeckGUI.cpp
    Created: 23 Jul 2023 12:16:53pm
    Author:  Ali

==============================================================================
*/
#include "AudioProcessorClass.h"
#include <JuceHeader.h>
#include "DeckGUI.h"


//==============================================================================
DeckGUI::DeckGUI(int _id,
                 DJAudioPlayer* _player,
                 juce::AudioFormatManager& formatManager,
                 juce::AudioThumbnailCache& thumbCache,AudioProcessorClass& audioProcessor
                ) : player(_player),
                    id(_id),
                    waveformDisplay(id, formatManager, thumbCache),
    audioProcessorClass(audioProcessor)
{
    // add all components and make visible
    addAndMakeVisible(playButton);
    addAndMakeVisible(stopButton);
    addAndMakeVisible(loadButton);
    addAndMakeVisible(volSlider);
    addAndMakeVisible(volLabel);
    addAndMakeVisible(speedSlider);
    addAndMakeVisible(speedLabel);
    addAndMakeVisible(posSlider);
    addAndMakeVisible(posLabel);
    addAndMakeVisible(reverbPlot1);
    addAndMakeVisible(reverbPlot2);
    addAndMakeVisible(waveformDisplay);

    //filter sliders
    lowPassSlider.setSliderStyle(juce::Slider::Rotary);
    lowPassSlider.setRange(20.0, 20000.0, 0.1);
    lowPassSlider.setValue(20000.0); // Initial value
    lowPassSlider.addListener(this);

    bandPassSlider.setSliderStyle(juce::Slider::Rotary);
    bandPassSlider.setRange(20.0, 20000.0, 0.1);
```

```
    bandPassSlider.setValue(1000.0); // Initial value
    bandPassSlider.addListener(this);

    highPassSlider.setSliderStyle(juce::Slider::Rotary);
    highPassSlider.setRange(20.0, 20000.0, 0.1);
    highPassSlider.setValue(20.0); // Initial value
    highPassSlider.addListener(this);



    addAndMakeVisible(&lowPassSlider);
    addAndMakeVisible(&bandPassSlider);
    addAndMakeVisible(&highPassSlider);


    // add listeners
    playButton.addListener(this);
    stopButton.addListener(this);
    loadButton.addListener(this);
    volSlider.addListener(this);
    speedSlider.addListener(this);
    posSlider.addListener(this);
    reverbSlider.addListener(this);
    reverbPlot1.addListener(this);
    reverbPlot2.addListener(this);

    //custom looks on components
    playButton.setLookAndFeel(&customLookAndFeel);
    stopButton.setLookAndFeel(&customLookAndFeel);
    loadButton.setLookAndFeel(&customLookAndFeel);
    volSlider.setLookAndFeel(&customLookAndFeel);
    speedSlider.setLookAndFeel(&customLookAndFeel);
    posSlider.setLookAndFeel(&customLookAndFeel);
    reverbSlider.setLookAndFeel(&customLookAndFeel);
    reverbPlot1.setLookAndFeel(&customLookAndFeel);
    reverbPlot2.setLookAndFeel(&customLookAndFeel);
    waveformDisplay.setLookAndFeel(&customLookAndFeel);

    //configure volume slider and label
    double volDefaultValue = 0.5;
    volSlider.setRange(0.0, 1.0);
    volSlider.setNumDecimalPlacesToDisplay(2);
    volSlider.setTextBoxStyle(juce::Slider::TextBoxLeft,
                              false,
                              50,
                              volSlider.getTextBoxHeight()
                             );
    volSlider.setValue(volDefaultValue);
    volSlider.setSkewFactorFromMidPoint(volDefaultValue);
    volLabel.setText("Volume", juce::dontSendNotification);
    volLabel.attachToComponent(&volSlider, true);

    //configure speed slider and label
```

```
    double speedDefaultValue = 1.0;
    speedSlider.setRange(0.25, 4.0); //reaches breakpoint if sliderValue ==
      0
    speedSlider.setNumDecimalPlacesToDisplay(2);
    speedSlider.setTextBoxStyle(juce::Slider::TextBoxLeft,
                                false,
                                50,
                                speedSlider.getTextBoxHeight()
                               );
    speedSlider.setValue(speedDefaultValue);
    speedSlider.setSkewFactorFromMidPoint(speedDefaultValue);
    speedLabel.setText("Speed", juce::dontSendNotification);
    speedLabel.attachToComponent(&speedSlider, true);

    //configure position slider and label
    posSlider.setRange(0.0, 1.0);
    posSlider.setNumDecimalPlacesToDisplay(2);
    posSlider.setTextBoxStyle(juce::Slider::TextBoxLeft,
                                false,
                                50,
                                posSlider.getTextBoxHeight()
                               );
    posLabel.setText("Position", juce::dontSendNotification);
    posLabel.attachToComponent(&posSlider, true);

    //configure reverb slider
    reverbSlider.setRange(0.0, 1.0);
    reverbSlider.setNumDecimalPlacesToDisplay(2);

    //configure reverb plots

    reverbPlot1.setTooltip("Set reverbe");
    reverbPlot2.setTooltip("Set reverbe");

    reverbPlot1.setLabelText("", "x: damping\ny: room size");
    reverbPlot2.setLabelText("", "x: dry level\ny: wet level");


    waveformDisplay.onPositionChanged = [this](double position) {
        player->setPositionRelative(position);
    };

    startTimer(500);
}

DeckGUI::~DeckGUI()
{
    stopTimer();
    playButton.setLookAndFeel(nullptr);
    stopButton.setLookAndFeel(nullptr);
    loadButton.setLookAndFeel(nullptr);
    volSlider.setLookAndFeel(nullptr);
    speedSlider.setLookAndFeel(nullptr);
```

```
    posSlider.setLookAndFeel(nullptr);
    reverbSlider.setLookAndFeel(nullptr);
    reverbPlot1.setLookAndFeel(nullptr);
    reverbPlot2.setLookAndFeel(nullptr);
    waveformDisplay.setLookAndFeel(nullptr);

}

void DeckGUI::paint (juce::Graphics& g)
{
    /* This demo code just fills the component's background and
       draws some placeholder text to get you started.

       You should replace everything in this method with your own
       drawing code..
    */


    // Calculate the x-coordinates for drawing lines
    int startX = volSlider.getX();
    int endX = volSlider.getX() + volSlider.getWidth();

    // Draw separator lines
    g.setColour(juce::Colours::black);  // Set color for the separator line


    g.fillAll (getLookAndFeel().findColour                                  ↵
      (juce::ResizableWindow::backgroundColourId));   // clear the         ↵
      background

    g.setColour (juce::Colours::transparentBlack);
    g.drawRect (getLocalBounds(), 1);   // draw an outline around the       ↵
      component
}

void DeckGUI::resized()
{
    /*This method is where you should set the bounds of any child
    components that your component contains..*/

    auto sliderLeft = getWidth() / 9;
    auto mainRight = getWidth() - getHeight() / 2;
    auto plotRight = getWidth() - mainRight;

    int buttonHeight = getHeight() / 8;

    //                    x start, y start, width, height
    playButton.setBounds(0, 0, mainRight / 3, buttonHeight);
    stopButton.setBounds(mainRight / 3, 0, mainRight / 3, buttonHeight);
    loadButton.setBounds(2 * mainRight / 3, 0, mainRight / 3, buttonHeight);

    lowPassSlider.setBounds(0, buttonHeight, mainRight / 3, buttonHeight);
    bandPassSlider.setBounds(mainRight / 3, buttonHeight, mainRight / 3,    ↵
```

```
      buttonHeight);
    highPassSlider.setBounds(2 * mainRight / 3, buttonHeight, mainRight / 3,
       buttonHeight);

    // Increasing the height of the sliders below to use up the space left
      by removed toggle buttons
    volSlider.setBounds(sliderLeft, 2 * buttonHeight, mainRight -
      sliderLeft, buttonHeight * 1.5);
    speedSlider.setBounds(sliderLeft, 3.5 * buttonHeight, mainRight -
      sliderLeft, buttonHeight * 1.5);
    posSlider.setBounds(sliderLeft, 5 * buttonHeight, mainRight -
      sliderLeft, buttonHeight * 1.5);

    reverbPlot1.setBounds(mainRight, 0, plotRight, getHeight() / 2);
    reverbPlot2.setBounds(mainRight, getHeight() / 2, plotRight, getHeight
      () / 2);

    // Expanding waveform display to use up the remaining space
    waveformDisplay.setBounds(0, 6.5 * buttonHeight, mainRight, 1.5 *
      buttonHeight);
}


//to handle button clicks
void DeckGUI::buttonClicked(juce::Button* button)
{
    if (button == &playButton)
    {
        DBG("Play button was clicked ");
        player->play();
    }
    if (button == &stopButton)
    {
        DBG("Stop button was clicked ");
        player->stop();
    }
    if (button == &loadButton)
    {
        DBG("Load button was clicked ");
        juce::FileChooser chooser{"Select a file"};
        if (chooser.browseForFileToOpen())
        {
            loadFile(juce::URL{ chooser.getResult() });
        }
    }
}

//to handle the slider value changes
void DeckGUI::sliderValueChanged(juce::Slider* sliderP)
{
    if (sliderP == &volSlider)
    {
        DBG("Volume slider moved " << sliderP->getValue());
```

```
        player->setGain(sliderP->getValue());
    }
    if (sliderP == &speedSlider)
    {
        DBG("Speed slider moved " << sliderP->getValue());
        player->setSpeed(sliderP->getValue());
    }
    if (sliderP == &posSlider)
    {
        DBG("Position slider moved " << sliderP->getValue());
        player->setPositionRelative(sliderP->getValue());
    }
    if (sliderP == &lowPassSlider)
    {
        DBG("Low Pass slider moved " << sliderP->getValue());
        player->getAudioProcessor().setLowPassFrequency(sliderP->getValue
          ());
    }
    if (sliderP == &bandPassSlider)
    {
        DBG("Band Pass slider moved " << sliderP->getValue());
        player->getAudioProcessor().setBandPassFrequency(sliderP->getValue
          ());
    }
    if (sliderP == &highPassSlider)
    {
        DBG("High Pass slider moved " << sliderP->getValue());
        player->getAudioProcessor().setHighPassFrequency(sliderP->getValue
          ());
    }

}

void DeckGUI::coordPlotValueChanged(CoordinatePlot* coordinatePlot)
{
    DBG("DeckGUI::coordPlotValueChanged called");
    if (coordinatePlot == &reverbPlot1)
    {
        DBG("Deck " << id << ": ReverbPlot1 was clicked");
        player->setRoomSize(coordinatePlot->getY());
        player->setDamping(coordinatePlot->getX());
    }
    if (coordinatePlot == &reverbPlot2)
    {
        DBG("Deck " << id << ": ReverbPlot2 was clicked");
        player->setWetLevel(coordinatePlot->getY());
        player->setDryLevel(coordinatePlot->getX());
    }
}

bool DeckGUI::isInterestedInFileDrag(const juce::StringArray& files)
{
    DBG("DeckGUI::isInterestedInFileDrag called. "
```

```
        + std::to_string(files.size()) + " file(s) being dragged.");
    return true;
}

void DeckGUI::filesDropped(const juce::StringArray& files, int x, int y)
{

    DBG("DeckGUI::filesDropped at " + std::to_string(x)
        + "x and " + std::to_string(y) + "y" );
    if (files.size() == 1)
    {
        loadFile(juce::URL{ juce::File{files[0]} });
    }
}

void DeckGUI::loadFile(juce::URL audioURL)
{
    DBG("DeckGUI::loadFile called");
    player->loadURL(audioURL);
    waveformDisplay.loadURL(audioURL);


}


void DeckGUI::timerCallback()
{
    //check if the relative position is greater than 0
    //otherwise loading file causes error
    if (player->getPositionRelative() > 0)
    {
        waveformDisplay.setPositionRelative(player->getPositionRelative());
    }
}


void DeckGUI::toggleLowPassFilter()
{
    lowPassEnabled = !lowPassEnabled;
}

void DeckGUI::toggleBandPassFilter()
{
    bandPassEnabled = !bandPassEnabled;
}

void DeckGUI::toggleHighPassFilter()
{
    highPassEnabled = !highPassEnabled;
}


void DeckGUI::setDJAudioPlayer(DJAudioPlayer* playerInstance)
```

```
{
    player = playerInstance;
}
```

```
/*
==============================================================================

    DeckGUI.h
    Created: 23 Jul 2023 12:16:53pm
    Author:  Ali

==============================================================================
*/

#pragma once

#include <JuceHeader.h>
#include "DJAudioPlayer.h"
#include "WaveformDisplay.h"
#include "CoordinatePlot.h"
#include "CustomLookAndFeel.h"
#include "AudioProcessorClass.h"

//==============================================================================
/*
*/
class DeckGUI  : public juce::Component,
                 public juce::Button::Listener,
                 public juce::Slider::Listener,
                 public juce::FileDragAndDropTarget,
                 public juce::Timer,
                 public CoordinatePlot::Listener
{
public:
    DeckGUI(int _id,
            DJAudioPlayer* player,
            juce::AudioFormatManager& formatManager,
            juce::AudioThumbnailCache& thumbCache, AudioProcessorClass&
              audioProcessor);
    ~DeckGUI() override;

    void paint (juce::Graphics&) override;
    void resized() override;

    /**Implement Button::Listener*/
    void buttonClicked(juce::Button* button) override;
    /**Implement Slider::Listener */
    void sliderValueChanged(juce::Slider* slider) override;
    /**Implement CoordinatePlot::Listener */
    void coordPlotValueChanged(CoordinatePlot* coordinatePlot) override;
    /**Detects if file is being dragged over deck*/
```

```
    bool isInterestedInFileDrag(const juce::StringArray& files) override;
    /**Detects if file is dropped onto deck*/
    void filesDropped(const juce::StringArray &files, int x, int y)
      override;
    /**Listen for changes to the waveform*/
    void timerCallback() override;

    bool isLowPassEnabled() const { return lowPassEnabled; }
    bool isBandPassEnabled() const { return bandPassEnabled; }
    bool isHighPassEnabled() const { return highPassEnabled; }

    void toggleLowPassFilter();
    void toggleBandPassFilter();
    void toggleHighPassFilter();


    void setDJAudioPlayer(DJAudioPlayer* playerInstance);

private:
    int id;

    juce::TextButton playButton{ "PLAY" };
    juce::TextButton stopButton{ "STOP" };
    juce::TextButton loadButton{ "LOAD" };
    juce::Slider volSlider;
    juce::Label volLabel;
    juce::Slider speedSlider;
    juce::Label speedLabel;
    juce::Slider posSlider;
    juce::Label posLabel;
    juce::Slider reverbSlider;
    juce::Slider slider;
    CoordinatePlot reverbPlot1;
    CoordinatePlot reverbPlot2;

    juce::Slider lowPassSlider;
    juce::Slider bandPassSlider;
    juce::Slider highPassSlider;

    juce::ToggleButton lowPassButton;
    juce::ToggleButton bandPassButton;
    juce::ToggleButton highPassButton;


    void loadFile(juce::URL audioURL);

    DJAudioPlayer* player;
    WaveformDisplay waveformDisplay;
    juce::SharedResourcePointer< juce::TooltipWindow > sharedTooltip;

    friend class PlaylistComponent;

    CustomLookAndFeel customLookAndFeel;
```

```
    AudioProcessorClass& audioProcessorClass;
    bool lowPassEnabled = true;
    bool bandPassEnabled = false;
    bool highPassEnabled = false;



    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(DeckGUI)


        int margin = 10;  // The space between sliders
    int lineThickness = 2;  // The thickness of separator lines

};
```

```cpp
/*
==============================================================================

    DJAudioPlayer.cpp
    Created: 23 Jul 2023 12:16:53pm
    Author:  Ali

==============================================================================
*/

#include "DJAudioPlayer.h"

// Constructor: initializes the format manager, sets default reverb settings, and prepares the audio processor
// Inputs: Reference to an existing AudioFormatManager instance
DJAudioPlayer::DJAudioPlayer(juce::AudioFormatManager& _formatManager) : formatManager(_formatManager)
{
    // (Self-written code) Set up the initial reverb parameters
    reverbParameters.roomSize = 0;
    reverbParameters.damping = 0;
    reverbParameters.wetLevel = 0;
    reverbParameters.dryLevel = 1.0;
    reverbSource.setParameters(reverbParameters);

    // (Self-written code) Initialize the audio processor with initial sample rate and block size
    double initialSampleRate = 44100.0;
    int initialSamplesPerBlock = 512;
    audioProcessor.prepareToPlay(initialSampleRate, initialSamplesPerBlock);
}

// Destructor: clean up resources when the instance is destroyed
DJAudioPlayer::~DJAudioPlayer()
{
}

// Prepares various sources for playback with given sample rate and block size
// Inputs: Expected samples per block, Sample rate
void DJAudioPlayer::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{
    transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    reverbSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    audioProcessor.prepareToPlay(sampleRate, samplesPerBlockExpected);
}
```

```cpp
// Processes the next block of audio
// Inputs: Information about the buffer to fill
void DJAudioPlayer::getNextAudioBlock(const juce::AudioSourceChannelInfo&
  bufferToFill)
{
    reverbSource.getNextAudioBlock(bufferToFill);
    audioProcessor.processAudioBlock(*bufferToFill.buffer);
}

// Releases resources allocated by various sources
void DJAudioPlayer::releaseResources()
{
    transportSource.releaseResources();
    resampleSource.releaseResources();
    reverbSource.releaseResources();
}

// Loads audio from a URL into the transport source
// Inputs: The URL of the audio to load
void DJAudioPlayer::loadURL(juce::URL audioURL)
{
    DBG("DJAudioPlayer::loadURL called");
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream
      (false));

    // (Self-written code) Load the reader into the transport source if
      valid
    if (reader != nullptr)
    {
        std::unique_ptr<juce::AudioFormatReaderSource> newSource(new
          juce::AudioFormatReaderSource(reader, true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader-
          >sampleRate);
        readerSource.reset(newSource.release());
    }
}

// Other methods follow a similar structure: simple, self-explanatory one-
  liners (self-written) with some debug information and parameter
  validation.

void DJAudioPlayer::play() { transportSource.start(); }
void DJAudioPlayer::stop() { transportSource.stop(); }
void DJAudioPlayer::setPosition(double posInSecs)
  { transportSource.setPosition(posInSecs); }

// A method to set the position relative to the length of the track
// Inputs: The relative position (between 0 and 1)
void DJAudioPlayer::setPositionRelative(double pos)
{
    // (Self-written code) Parameter validation and conversion to seconds
    if (pos < 0 || pos > 1.0)
    {
```

```
        DBG("DJAudioPlayer::setPositionRelative position should be between 0 and 1");
    }
    else
    {
        double posInSecs = transportSource.getLengthInSeconds() * pos;
        setPosition(posInSecs);
    }
}

// (Self-written code) Below methods are similar, setting various parameters with some validation. They change aspects such as gain, speed, and reverb settings.

void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
    }
    else { transportSource.setGain(gain); }
}

void DJAudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0.25 || ratio > 4.0)
    {
        DBG("DJAudioPlayer::setSpeed ratio should be between 0.25 and 4");
    }
    else { resampleSource.setResamplingRatio(ratio); }
}

void DJAudioPlayer::setRoomSize(float size)
{
    DBG("DJAudioPlayer::setRoomSize called");
    if (size < 0 || size > 1.0)
    {
        DBG("DJAudioPlayer::setRoomSize size should be between 0 and 1.0");
    }
    else
    {
        reverbParameters.roomSize = size;
        reverbSource.setParameters(reverbParameters);
    }
}

void DJAudioPlayer::setDamping(float dampingAmt)
{
    DBG("DJAudioPlayer::setDamping called");
    if (dampingAmt < 0 || dampingAmt > 1.0)
    {
        DBG("DJAudioPlayer::setDamping amount should be between 0 and 1.0");
    }
```

```
    else
    {
        reverbParameters.damping = dampingAmt;
        reverbSource.setParameters(reverbParameters);
    }
}

void DJAudioPlayer::setWetLevel(float wetLevel)
{
    DBG("DJAudioPlayer::setWetLevel called");
    if (wetLevel < 0 || wetLevel > 1.0)
    {
        DBG("DJAudioPlayer::setWetLevel level should be between 0 and 1.0");
    }
    else
    {
        reverbParameters.wetLevel = wetLevel;
        reverbSource.setParameters(reverbParameters);
    }
}

void DJAudioPlayer::setDryLevel(float dryLevel)
{
    DBG("DJAudioPlayer::setDryLevel called");
    if (dryLevel < 0 || dryLevel > 1.0)
    {
        DBG("DJAudioPlayer::setDryLevel level should be between 0 and 1.0");
    }
    else
    {
        reverbParameters.dryLevel = dryLevel;
        reverbSource.setParameters(reverbParameters);
    }
}

// Returns the current position relative to the length of the track
// Outputs: The relative position as a double
double DJAudioPlayer::getPositionRelative()
{
    return transportSource.getCurrentPosition() /
      transportSource.getLengthInSeconds();
}

// Returns the length of the current track in seconds
// Outputs: The length in seconds as a double
double DJAudioPlayer::getLengthInSeconds()
{
    return transportSource.getLengthInSeconds();
}

// Provides access to the internal audio processor instance
// Outputs: Reference to the internal audio processor instance
AudioProcessorClass& DJAudioPlayer::getAudioProcessor()
```

```
{
    return audioProcessor;
}
```

```
/*
==============================================================================

    DJAudioPlayer.h
    Created: 22 Jul 2023 8:28:03pm
    Author:  Ali

==============================================================================
*/

#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
#include "AudioProcessorClass.h"


class DJAudioPlayer : public juce::AudioSource
{
    public:
        DJAudioPlayer(juce::AudioFormatManager& _formatManager);
        ~DJAudioPlayer();

        void prepareToPlay(int samplesPerBlockExpected, double sampleRate) override;
        void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill) override;
        void releaseResources() override;

        /**Loads the audio file*/
        void loadURL(juce::URL audioURL);
        /**Plays loaded audio file*/
        void play();
        /**Stops playing audio file*/
        void stop();
        /**Sets relative position of audio file*/
        void setPositionRelative(double pos);
        /**Sets the volume*/
        void setGain(double gain);
        /**Sets the speed*/
        void setSpeed(double ratio);
        /**Gets relative position of playhead*/
        double getPositionRelative();
        /**Gets the length of transport source in seconds*/
        double getLengthInSeconds();
        /**Sets the amount of reverb*/
        void setRoomSize(float size);
        /**Sets the amount of reverb*/
        void setDamping(float dampingAmt);
        /**Sets the amount of reverb*/
        void setWetLevel(float wetLevel);
```

```
        /**Sets the amount of reverb*/
        void setDryLevel(float dryLevel);

        AudioProcessorClass& getAudioProcessor();
    private:
        void setPosition(double posInSecs);
        juce::AudioFormatManager& formatManager;
        std::unique_ptr<juce::AudioFormatReaderSource> readerSource;
        juce::AudioTransportSource transportSource;
        juce::ResamplingAudioSource resampleSource{ &transportSource, false, ↵
           2 };
        juce::ReverbAudioSource reverbSource{ &resampleSource, false };
        juce::Reverb::Parameters reverbParameters;

        AudioProcessorClass audioProcessor;
};
```

```cpp
/*
==============================================================================

    This file contains the basic startup code for a JUCE application.

==============================================================================
*/

#include <JuceHeader.h>
#include "MainComponent.h"

//==============================================================================
class OtoDecksApplication  : public juce::JUCEApplication
{
public:
    //==============================================================================
    OtoDecksApplication() {}

    const juce::String getApplicationName() override       { return ProjectInfo::projectName; }
    const juce::String getApplicationVersion() override    { return ProjectInfo::versionString; }
    bool moreThanOneInstanceAllowed() override             { return true; }

    //==============================================================================
    void initialise (const juce::String& commandLine) override
    {
        // This method is where you should put your application's initialisation code..

        mainWindow.reset (new MainWindow (getApplicationName()));
    }

    void shutdown() override
    {
        // Add your application's shutdown code here..

        mainWindow = nullptr; // (deletes our window)
    }

    //==============================================================================
```

```
    void systemRequestedQuit() override
    {
        // This is called when the app is being asked to quit: you can
          ignore this
        // request and let the app carry on running, or call quit() to allow
           the app to close.
        quit();
    }

    void anotherInstanceStarted (const juce::String& commandLine) override
    {
        // When another instance of the app is launched while this one is
          running,
        // this method is invoked, and the commandLine parameter tells you
          what
        // the other instance's command-line arguments were.
    }

    //
      ======================================================================
      ========
    /*
        This class implements the desktop window that contains an instance
          of
        our MainComponent class.
    */
    class MainWindow    : public juce::DocumentWindow
    {
    public:
        MainWindow (juce::String name)
            : DocumentWindow (name,
                              juce::Desktop::getInstance
                    ().getDefaultLookAndFeel()
                                                        .findColour
                    (juce::ResizableWindow::backgroundColourId),
                              DocumentWindow::allButtons)
        {
            setUsingNativeTitleBar (true);
            setContentOwned (new MainComponent(), true);

           #if JUCE_IOS || JUCE_ANDROID
            setFullScreen (true);
           #else
            setResizable (true, true);
            centreWithSize (getWidth(), getHeight());
           #endif

            setVisible (true);
        }

        void closeButtonPressed() override
        {
            // This is called when the user tries to close this window.
```

```
              Here, we'll just
            // ask the app to quit when this happens, but you can change
              this to do
            // whatever you need.
            JUCEApplication::getInstance()->systemRequestedQuit();
        }

        /* Note: Be careful if you override any DocumentWindow methods - the
           base
           class uses a lot of them, so by overriding you might break its
             functionality.
           It's best to do all your work in your content component instead,
             but if
           you really have to override any DocumentWindow methods, make sure
              your
           subclass also calls the superclass's method.
        */

    private:
        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainWindow)
    };

private:
    std::unique_ptr<MainWindow> mainWindow;
};

//
  ==========================================================================
  ====
// This macro generates the main() routine that launches the app.
START_JUCE_APPLICATION (OtoDecksApplication)
```

```cpp
#include "MainComponent.h"

//
  ==============================================================================
  ====
MainComponent::MainComponent()
{
    // Make sure you set the size of the component after
    // you add any child components.
    setSize (944, 600);

    // Some platforms require permissions to open input channels so request
      that here
    if (juce::RuntimePermissions::isRequired
      (juce::RuntimePermissions::recordAudio)
        && ! juce::RuntimePermissions::isGranted
          (juce::RuntimePermissions::recordAudio))
    {
        juce::RuntimePermissions::request
          (juce::RuntimePermissions::recordAudio,
                                          [&] (bool granted)
                    { setAudioChannels (granted ? 2 : 0, 2); });
    }
    else
    {
        // Specify the number of input and output channels that we want to
          open
        setAudioChannels (2, 2);
    }

    addAndMakeVisible(deckGUI1);
    addAndMakeVisible(deckGUI2);
    addAndMakeVisible(playlistComponent);

    formatManager.registerBasicFormats();
}

MainComponent::~MainComponent()
{
    // This shuts down the audio device and clears the audio source.
    shutdownAudio();
}

//
  ==============================================================================
  ====
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double
  sampleRate)
{
    // This function will be called when the audio device is started, or
      when
    // its settings (i.e. sample rate, block size, etc) are changed.
```

```
    // You can use this function to initialise any resources you might need,
    // but be careful - it will be called on the audio thread, not the GUI
      thread.

    // For more details, see the help for AudioProcessor::prepareToPlay()

    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
    player2.prepareToPlay(samplesPerBlockExpected, sampleRate);

}
void MainComponent::getNextAudioBlock(const juce::AudioSourceChannelInfo&
  bufferToFill)
{
    mixerSource.getNextAudioBlock(bufferToFill);
}

void MainComponent::releaseResources()
{
    // This will be called when the audio device stops, or when it is being
    // restarted due to a setting change.

    // For more details, see the help for AudioProcessor::releaseResources()
    mixerSource.removeAllInputs();
    mixerSource.releaseResources();
    player1.releaseResources();
    player2.releaseResources();
}

//
  ==============================================================================
  ====
void MainComponent::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background
      with a solid colour)
    g.fillAll (getLookAndFeel().findColour
      (juce::ResizableWindow::backgroundColourId));

    // You can add your drawing code here!
}

void MainComponent::resized()
{
    // This is called when the MainContentComponent is resized.
    // If you add any child components, this is where you should
    // update their positions.

    //playlistComponent.setBounds(0, 0, getWidth() / 3, getHeight());
    //deckGUI1.setBounds(getWidth() / 3, 0, 2 * getWidth() / 3, getHeight
      () / 2);
    //deckGUI2.setBounds(getWidth() / 3, getHeight() / 2, 2 * getWidth() /
```

```
      3, getHeight() / 2);
    int columns = 100;
    auto playlistRight = 28 * getWidth() / columns;
    playlistComponent.setBounds(0, 0, playlistRight, getHeight());
    deckGUI1.setBounds(playlistRight, 0, getWidth() - playlistRight,
      getHeight() / 2);
    deckGUI2.setBounds(playlistRight, getHeight() / 2, getWidth() -
      playlistRight, getHeight() / 2);

    //getWidth() - getWidth() / columns - getHeight() / 4
    //deckGUI1.setBounds(playlistRight, 0, getWidth() - playlistRight -
      getHeight() / 4, getHeight() / 2);
    //deckGUI2.setBounds(playlistRight, getHeight() / 2, getWidth() -
      playlistRight - getHeight() / 4, getHeight() / 2);
}
```

```
#pragma once

#include <JuceHeader.h>
#include <juce_gui_basics\juce_gui_basics.h>
#include "DJAudioPlayer.h"
#include "DeckGUI.h"
#include "PlaylistComponent.h"
#include "AudioProcessorClass.h"


//
  ==============================================================================
  ====
/*
    This component lives inside our window, and this is where you should put
       all
    your controls and content.
*/
class MainComponent  : public juce::AudioAppComponent
{
public:
    //
      ==============================================================================
      ========
    MainComponent();
    ~MainComponent() override;

    //
      ==============================================================================
      ========
    void prepareToPlay (int samplesPerBlockExpected, double sampleRate)
      override;
    void getNextAudioBlock (const juce::AudioSourceChannelInfo&
      bufferToFill) override;
    void releaseResources() override;

    //
      ==============================================================================
      ========
    void paint (juce::Graphics& g) override;
    void resized() override;

private:
    //
      ==============================================================================
      ========
    // Your private member variables go here...

    juce::AudioFormatManager formatManager;
    juce::AudioThumbnailCache thumbCache{100};

    AudioProcessorClass audioProcessor;
```

```
    DJAudioPlayer player1{formatManager};
    DJAudioPlayer player2{formatManager};
    DJAudioPlayer playerForParsingMetaData{formatManager};
    DeckGUI deckGUI1{1, &player1, formatManager,
      thumbCache,audioProcessor };
    DeckGUI deckGUI2{2, &player2, formatManager,
      thumbCache,audioProcessor };
    PlaylistComponent playlistComponent{ &deckGUI1, &deckGUI2,
      &playerForParsingMetaData };

    juce::MixerAudioSource mixerSource;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};
```

```cpp
/*
==============================================================================
    PlaylistComponent.cpp
    Created: 24 Jul 2023 3:29:26pm
    Author:  Ali
==============================================================================
*/

#include <JuceHeader.h>
#include "PlaylistComponent.h"
#include "AlertCallback.h"

//==============================================================================
PlaylistComponent::PlaylistComponent(DeckGUI* _deckGUI1,
    DeckGUI* _deckGUI2,
    DJAudioPlayer* _playerForParsingMetaData
)
    : deckGUI1(_deckGUI1),
    deckGUI2(_deckGUI2),
    playerForParsingMetaData(_playerForParsingMetaData)
{
    // Child components and initial settings setup (Self-written section)
    addAndMakeVisible(importButton);
    addAndMakeVisible(searchField);
    addAndMakeVisible(library);
    addAndMakeVisible(addToPlayer1Button);
    addAndMakeVisible(addToPlayer2Button);

    importButton.addListener(this);
    searchField.addListener(this);
    addToPlayer1Button.addListener(this);
    addToPlayer2Button.addListener(this);

    searchField.setTextToShowWhenEmpty("Search Tracks (enter to submit)", juce::Colours::orange);
    searchField.onReturnKey = [this] { searchLibrary(searchField.getText()); };

    // Table setup and library load (Self-written section)
    library.getHeader().addColumn("Tracks", 1, 1);
    library.getHeader().addColumn("Length", 2, 1);
    library.getHeader().addColumn("", 3, 1);
    library.setModel(this);
    loadLibrary();
}

PlaylistComponent::~PlaylistComponent()
```

```cpp
{
    // Self-written destructor
    saveLibrary();
}

void PlaylistComponent::alertWindowCallback(int returnValue)
{
    // Callback for alert window (Self-written section)
    if (returnValue == 1)
    {
        // Actions for the OK button can be defined here
    }
}

void PlaylistComponent::paint(juce::Graphics& g)
{
    // JUCE generated paint method, with custom drawing code added
    g.fillAll(getLookAndFeel().findColour
      (juce::ResizableWindow::backgroundColourId));
    g.setColour(juce::Colours::grey);
    g.drawRect(getLocalBounds(), 1);
    g.setColour(juce::Colours::white);
    g.setFont(14.0f);
}

void PlaylistComponent::resized()
{
    // Setting the bounds for each component (Self-written section)
    importButton.setBounds(0, 0, getWidth(), getHeight() / 16);
    library.setBounds(0, 1 * getHeight() / 16, getWidth(), 13 * getHeight
      () / 16);
    searchField.setBounds(0, 14 * getHeight() / 16, getWidth(), getHeight
      () / 16);
    addToPlayer1Button.setBounds(0, 15 * getHeight() / 16, getWidth() / 2,
      getHeight() / 16);
    addToPlayer2Button.setBounds(getWidth() / 2, 15 * getHeight() / 16,
      getWidth() / 2, getHeight() / 16);

    // Setting column widths (Self-written section)
    library.getHeader().setColumnWidth(1, 12.8 * getWidth() / 20);
    library.getHeader().setColumnWidth(2, 5 * getWidth() / 20);
    library.getHeader().setColumnWidth(3, 2 * getWidth() / 20);
}

int PlaylistComponent::getNumRows()
{
    // Return the number of tracks available (Self-written section)
    return tracks.size();
}

void PlaylistComponent::paintRowBackground(juce::Graphics& g, int rowNumber,
   int width, int height, bool rowIsSelected)
{
```

```
    // Setting row background colors (Self-written section)
    if (rowIsSelected)
    {
        g.fillAll(juce::Colours::orange);
    }
    else
    {
        g.fillAll(juce::Colours::darkgrey);
    }
}

void PlaylistComponent::paintCell(juce::Graphics& g, int rowNumber, int
  columnId, int width, int height, bool rowIsSelected)
{
    // Display track titles and lengths (Self-written section)
    if (rowNumber < getNumRows())
    {
        if (columnId == 1)
        {
            g.drawText(tracks[rowNumber].title, 2, 0, width - 4, height,
              juce::Justification::centredLeft, true);
        }
        if (columnId == 2)
        {
            g.drawText(tracks[rowNumber].length, 2, 0, width - 4, height,
              juce::Justification::centred, true);
        }
    }
}

juce::Component* PlaylistComponent::refreshComponentForCell(int rowNumber,
  int columnId, bool isRowSelected, Component* existingComponentToUpdate)
{
    // Creates delete buttons in each row (Self-written section)
    if (columnId == 3)
    {
        if (existingComponentToUpdate == nullptr)
        {
            juce::TextButton* btn = new juce::TextButton{ "X" };
            juce::String id{ std::to_string(rowNumber) };
            btn->setComponentID(id);
            btn->addListener(this);
            existingComponentToUpdate = btn;
        }
    }
    return existingComponentToUpdate;
}

// This function is a callback for when any button in the playlist component
   is clicked.
// Depending on the button clicked, it triggers different actions such as
  loading a track in a player or importing tracks to the library.
void PlaylistComponent::buttonClicked(juce::Button* button)
```

```
{
    // If the import button is clicked, logs the click, imports tracks to
      the library, and updates the library content.
    if (button == &importButton)
    {
        DBG("Load button clicked");
        importToLibrary();
        library.updateContent();
    }
    // If the add to player 1 button is clicked, logs the click and loads
      the selected track in player 1.
    else if (button == &addToPlayer1Button)
    {
        DBG("Add to Player 1 clicked");
        loadInPlayer(deckGUI1);
    }
    // If the add to player 2 button is clicked, logs the click and loads
      the selected track in player 2.
    else if (button == &addToPlayer2Button)
    {
        DBG("Add to Player 2 clicked");
        loadInPlayer(deckGUI2);
    }
    // For other buttons, retrieves the ID of the button clicked, logs the
      track removal and removes the track from the tracks vector, and
      updates the library content.
    else
    {
        int id = std::stoi(button->getComponentID().toStdString());
        DBG(tracks[id].title + " removed from Library");
        deleteFromTracks(id);
        library.updateContent();
    }
}

// This function handles the loading of a selected track into a specified
  player.
// If no track is selected, it displays an alert window with a message
  prompting the user to select a track.
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow{ library.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Adding: " << tracks[selectedRow].title << " to Player");
        deckGUI->loadFile(tracks[selectedRow].URL);
    }
    else
    {
        juce::AlertWindow::showMessageBoxAsync(juce::AlertWindow::InfoIcon,
            "Add to Deck Information",
            "Please select a track to add to deck",
            "OK",
```

```
            nullptr,
            new AlertCallback(this));
    }
}

void PlaylistComponent::importToLibrary()
{
    DBG("PlaylistComponent::importToLibrary called");

    // initialize file chooser
    juce::FileChooser chooser{ "Select files" };
    if (chooser.browseForMultipleFilesToOpen()) // Reverted to the correct method
    {
        for (const juce::File& file : chooser.getResults())
        {
            juce::String fileNameWithoutExtension { file.getFileNameWithoutExtension() };
            if (!isInTracks(fileNameWithoutExtension)) // if not already loaded
            {
                Track newTrack{ file };
                juce::URL audioURL{ file };
                newTrack.length = getLength(audioURL);
                tracks.push_back(newTrack);
                DBG("loaded file: " << newTrack.title);
            }
            else // display info message
            {
                if (juce::AlertWindow::showOkCancelBox (juce::AlertWindow::InfoIcon,
                    "Load information:",
                    fileNameWithoutExtension + " already loaded"))
                {
                    // OK button was clicked, you can put any action you want to happen here
                }
            }
        }
    }
}

bool PlaylistComponent::isInTracks(juce::String fileNameWithoutExtension)
{
    return (std::find(tracks.begin(), tracks.end(), fileNameWithoutExtension) != tracks.end());
}

void PlaylistComponent::deleteFromTracks(int id)
{
    tracks.erase(tracks.begin() + id);
}
```

```
juce::String PlaylistComponent::getLength(juce::URL audioURL)
{
    playerForParsingMetaData->loadURL(audioURL);
    double seconds{ playerForParsingMetaData->getLengthInSeconds() };
    juce::String minutes{ secondsToMinutes(seconds) };
    return minutes;
}

juce::String PlaylistComponent::secondsToMinutes(double seconds)
{
    //find seconds and minutes and make into string
    int secondsRounded{ int(std::round(seconds)) };
    juce::String min{ std::to_string(secondsRounded / 60) };
    juce::String sec{ std::to_string(secondsRounded % 60) };

    if (sec.length() < 2) // if seconds is 1 digit or less
    {
        //add '0' to seconds until seconds is length 2
        sec = sec.paddedLeft('0', 2);
    }
    return juce::String{ min + ":" + sec };
}

void PlaylistComponent::searchLibrary(juce::String searchText)
{
    DBG("Searching library for: " << searchText);
    if (searchText != "")
    {
        int rowNumber = whereInTracks(searchText);
        library.selectRow(rowNumber);
    }
    else
    {
        library.deselectAllRows();
    }
}

int PlaylistComponent::whereInTracks(juce::String searchText)
{
    // finds index where track title contains searchText
    auto it = find_if(tracks.begin(), tracks.end(),
        [&searchText](const Track& obj) {return obj.title.contains
          (searchText); });
    int i = -1;

    if (it != tracks.end())
    {
        i = std::distance(tracks.begin(), it);
    }

    return i;
}
```

```cpp
void PlaylistComponent::saveLibrary()
{
    // create .csv to save library
    std::ofstream myLibrary("my-library.csv");

    // save library to file
    for (Track& t : tracks)
    {
        myLibrary << t.file.getFullPathName() << "," << t.length << "\n";
    }
}

void PlaylistComponent::loadLibrary()
{
    // create input stream from saved library
    std::ifstream myLibrary("my-library.csv");
    std::string filePath;
    std::string length;

    // Read data, line by line
    if (myLibrary.is_open())
    {
        while (getline(myLibrary, filePath, ',')) {
            juce::File file{ filePath };
            Track newTrack{ file };

            getline(myLibrary, length);
            newTrack.length = length;
            tracks.push_back(newTrack);
        }
    }
    myLibrary.close();
}
```

```
/*
==============================================================================

    PlaylistComponent.h
    Created: 24 Jul 2023 3:29:26pm
    Author:  Ali

==============================================================================
*/

#pragma once

#include <JuceHeader.h>
#include <vector>
#include <algorithm>
#include <fstream>
#include "Track.h"
#include "DeckGUI.h"
#include "DJAudioPlayer.h"

//==============================================================================
/*
*/
class PlaylistComponent  : public juce::Component,
                           public juce::TableListBoxModel,
                           public juce::Button::Listener,
                           public juce::TextEditor::Listener
{
public:
    PlaylistComponent(DeckGUI* _deckGUI1,
                      DeckGUI* _deckGUI2,
                      DJAudioPlayer* _playerForParsingMetaData
                     );
    ~PlaylistComponent() override;

    void paint (juce::Graphics&) override;
    void resized() override;

    void alertWindowCallback(int returnValue);

    int getNumRows() override;
    void paintRowBackground(juce::Graphics& g,
                            int rowNumber,
                            int width,
                            int height,
                            bool rowIsSelected
                           ) override;
```

```
    void paintCell(juce::Graphics& g,
                   int rowNumber,
                   int columnId,
                   int width,
                   int height,
                   bool rowIsSelected
                  ) override;

    Component* refreshComponentForCell(int rowNumber,
                                       int columnId,
                                       bool isRowSelected,
                                       Component* existingComponentToUpdate)
                    override;
    void buttonClicked(juce::Button* button) override;
private:
    std::vector<Track> tracks;

    juce::TextButton importButton{ "IMPORT TRACKS" };
    juce::TextEditor searchField;
    juce::TableListBox library;
    juce::TextButton addToPlayer1Button{ "ADD TO DECK 1" };
    juce::TextButton addToPlayer2Button{ "ADD TO DECK 2" };

    DeckGUI* deckGUI1;
    DeckGUI* deckGUI2;
    DJAudioPlayer* playerForParsingMetaData;

    juce::String getLength(juce::URL audioURL);
    juce::String secondsToMinutes(double seconds);

    void importToLibrary();
    void searchLibrary(juce::String searchText);
    void saveLibrary();
    void loadLibrary();
    void deleteFromTracks(int id);
    bool isInTracks(juce::String fileNameWithoutExtension);
    int whereInTracks(juce::String searchText);
    void loadInPlayer(DeckGUI* deckGUI);

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (PlaylistComponent)
};
```

```cpp
/*
==============================================================================

    Track.cpp
    Created: 4 Aug 2023 10:16:10am
    Author:  Ali

==============================================================================
*/

#include "Track.h"
#include <filesystem>

// Constructor definition for the Track class.
// It initializes the file, title, and URL members of the class with the given file parameter.
// The title is derived from the file name without extension, and the URL is constructed from the file object.
// It also logs the creation of a new track.
Track::Track(juce::File _file)
    : file(std::move(_file)), // Moving the file object to avoid unnecessary copies
    title(file.getFileNameWithoutExtension()), // Initializing title with the filename without its extension
    URL(juce::URL{ file }) // Initializing URL with a juce::URL object created from the file
{
    // Logging the creation of a new track with its title
    DBG("Created new track with title: " << title);
}

// Overloading the equality operator to compare a Track object with a juce::String object (presumably a title).
// It returns true if the title member of the Track object is equal to the input juce::String object.
bool Track::operator==(const juce::String& other) const
{
    // Comparing the title member with the input string and returning the result
    return title == other;
}
```

```
/*
                                                                                ↵
==============================================================================↵
==

    Track.h
    Created: 4 Aug 2023 10:16:10am
    Author:  Ali

                                                                                ↵
==============================================================================↵
==
*/

#pragma once
#include <JuceHeader.h>

class Track
{
    public:
        Track(juce::File _file);
        juce::File file;
        juce::URL URL;
        juce::String title;
        juce::String length;
        /**objects are compared by title*/
        bool operator==(const juce::String& other) const;
};
```

```cpp
/*
==============================================================================

    WaveformDisplay.cpp
    Created: 24 Jul 2023 10:55:44am
    Author:  Ali

==============================================================================
*/

#include <JuceHeader.h>
#include "WaveformDisplay.h"


#include "CustomLookAndFeel.h"

//==============================================================================
WaveformDisplay::WaveformDisplay(int _id,
                                 juce::AudioFormatManager& formatManager,
                                 juce::AudioThumbnailCache& thumbCache
                                ) : audioThumb(1000, formatManager,
                    thumbCache),
                                    fileLoaded(false),
                                    position(0),
                                    id(_id)
{
    // In your constructor, you should add any child components, and
    // initialise any special settings that your component needs.
    setLookAndFeel(&customLookAndFeel);
    audioThumb.addChangeListener(this);
}
// Destructor which resets the look and feel
WaveformDisplay::~WaveformDisplay()
{
    setLookAndFeel(nullptr);
}
// Personal code: This function sets the waveform data and marks the component to be repainted.
void WaveformDisplay::setWaveformData(const std::vector<float>& data)
    {
        waveformData = data;
        dataSize = data.size();
        repaint();
    }

// The paint method which draws the component and the waveform
void WaveformDisplay::paint(juce::Graphics& g)
```

```
{
    // Existing setup code
    g.fillAll(getLookAndFeel().findColour
      (juce::ResizableWindow::backgroundColourId)); // clear the background
    g.setColour(juce::Colours::grey);
    g.drawRect(getLocalBounds(), 1);   // draw an outline around the
      component
    g.setColour(juce::Colours::hotpink);

    g.setColour(juce::Colours::orangered);
    g.setFont(18.0f);
    g.drawText("Deck: " + std::to_string(id), getLocalBounds(),
        juce::Justification::centredTop,true);

    if (fileLoaded)
    {
        // Existing drawing code
        g.setFont(15.0f);
        audioThumb.drawChannel(g,
            getLocalBounds(),
            0,
            audioThumb.getTotalLength(),
            0,
            1.0f
        );
        g.setColour(juce::Colours::lightgreen);
        g.drawRect(position * getWidth(), 0, getWidth() / 20, getHeight());
        g.setColour(juce::Colours::white);
        g.drawText(fileName, getLocalBounds(),
            juce::Justification::bottomLeft, true);

        // Drawing the waveform data from waveformData vector using
          CustomLookAndFeel
        if (!waveformData.empty())
        {
            customLookAndFeel.drawWaveform(g, getLocalBounds(),
              waveformData.data(), dataSize);
        }
    }
    else
    {
        // Existing file not loaded case
        g.setFont(20.0f);
        g.setColour(juce::Colours::green);

        g.drawText("File not loaded...", getLocalBounds(),
            juce::Justification::centred, true);   // draw some placeholder
              text
    }
}


void WaveformDisplay::resized()
```

```
{
    // This method is where you should set the bounds of any child
    // components that your component contains..

}
// Change listener callback that gets called when the audio thumbnail
  changes
void WaveformDisplay::changeListenerCallback(juce::ChangeBroadcaster*
  source)
{
    repaint();
}
// Method to load an audio URL and create a thumbnail for it
void WaveformDisplay::loadURL(juce::URL audioURL)
{
    DBG("WaveformDisplay::loadURL called");
    audioThumb.clear();
    fileLoaded = audioThumb.setSource(new juce::URLInputSource(audioURL));
    if (fileLoaded)
    {
        DBG("WaveformDisplay::loadURL file loaded");
        fileName = audioURL.getFileName();// Setting the filename to be
          displayed on the component (Personal code)
        repaint();// Repainting the component to show the loaded file
    }
    else
    {
        DBG("WaveformDisplay::loadURL file NOT loaded");
    }
}
// Method to set the relative position of a marker on the waveform (Personal
   code)
void WaveformDisplay::setPositionRelative(double pos)
{
    if (pos != position)
    {
        position = pos;
        repaint();// Repainting the component to update the position marker
    }
}
// Mouse down event handler for interacting with the waveform (Personal
  code)
void WaveformDisplay::mouseDown(const juce::MouseEvent& event)
{
    if (fileLoaded)
    {
        position = static_cast<double>(event.x) / getWidth();
        if (onPositionChanged)
        {
            onPositionChanged(position);
        }
        repaint();
    }
```

```
}


// Mouse drag event handler for dragging the position marker on the waveform
   (Personal code)
void WaveformDisplay::mouseDrag(const juce::MouseEvent& event)
{
    if (fileLoaded)
    {
        position = static_cast<double>(event.x) / getWidth();
        if (onPositionChanged)
        {
            onPositionChanged(position);
        }
        repaint();
    }
}
```

```cpp
/*
==============================================================================
    AlertCallback.cpp
    Created: 5 Sep 2023 9:58:39pm
    Author:  Ali
==============================================================================
*/

#include "AlertCallback.h"

/**
 * Constructor: AlertCallback (Self-generated)
 *
 * Purpose:
 *   Initializes a new instance of the AlertCallback class, associating it with a PlaylistComponent instance.
 *
 * Inputs:
 *   playlistComponent - A pointer to a PlaylistComponent instance. This will be used for invoking the alert window callback method.
 */
AlertCallback::AlertCallback(PlaylistComponent* playlistComponent)
    : playlistComponent(playlistComponent)
{
    // Constructor body: Initialization of playlistComponent member variable (Self-generated)
}

/**
 * Function: modalStateFinished (Self-generated)
 *
 * Purpose:
 *   This function is invoked when a modal state has ended. It triggers the alert window callback method in the associated PlaylistComponent instance.
 *
 * Inputs:
 *   result - An integer representing the result of the modal state (e.g., which button was pressed in a modal dialog).
 */
void AlertCallback::modalStateFinished(int result)
{
    // Triggering the alert window callback with the modal state result (Self-generated)
    playlistComponent->alertWindowCallback(result);
}
```

```cpp
/*
==============================================================================
    AlertCallback.h
    Created: 5 Sep 2023 9:58:20pm
    Author:  Ali
==============================================================================
*/

#pragma once
#include <JuceHeader.h>
#include "PlaylistComponent.h" // Include the header file for PlaylistComponent (Self-generated)

// Class: AlertCallback (Self-generated)
// Extends the juce::ModalComponentManager::Callback to handle modal state transitions.
// It holds a reference to a PlaylistComponent to perform actions when the modal state is finished.
class AlertCallback : public juce::ModalComponentManager::Callback
{
public:
    // Constructor: AlertCallback (Self-generated)
    // Purpose:
    //   Initializes an instance of the AlertCallback class.
    // Inputs:
    //   playlistComponent - A pointer to the PlaylistComponent instance that this AlertCallback will interact with.
    AlertCallback(PlaylistComponent* playlistComponent);

    // Function: modalStateFinished (Self-generated)
    // Purpose:
    //   Defines the actions to be performed when the modal state finishes.
    //   Specifically, it calls the alertWindowCallback function of the associated PlaylistComponent.
    // Inputs:
    //   result - An integer representing the result of the modal state (e.g., the button pressed in a modal dialog).
    void modalStateFinished(int result) override;

private:
    // Member Variable: playlistComponent (Self-generated)
    // Holds a pointer to the PlaylistComponent instance to interact with when the modal state finishes.
    PlaylistComponent* playlistComponent;
};
```

```cpp
/*
==============================================================================
    AudioProcessorClass.cpp
    Created: 10 Sep 2023 12:29:23pm
    Author:  Ali
==============================================================================
*/

#include "AudioProcessorClass.h"
#include "JuceHeader.h"

// Constructor: Initializes the AudioProcessorClass with a default sample rate (Self-written)
AudioProcessorClass::AudioProcessorClass()
    : lastSampleRate(44100.0), currentSampleRate(44100.0)
{
}

// Destructor: Cleans up the AudioProcessorClass instance (Self-written)
AudioProcessorClass::~AudioProcessorClass()
{
}

// Function: prepareToPlay (Self-written)
// Purpose: Prepares the audio processor to play, setting up necessary filters with the initial configurations.
// Inputs:
//   sampleRate - The sample rate for the audio stream.
//   samplesPerBlock - The number of samples per block.
void AudioProcessorClass::prepareToPlay(double sampleRate, int samplesPerBlock)
{
    lastSampleRate = sampleRate;
    currentSampleRate = sampleRate;

    juce::dsp::ProcessSpec spec { sampleRate, static_cast<uint32_t>(samplesPerBlock), 2 };

    lowPassFilter.prepare(spec);
    lowPassFilter.coefficients = juce::dsp::IIR::Coefficients<float>::makeLowPass(sampleRate, 20000.0f);

    bandPassFilter.prepare(spec);
    bandPassFilter.coefficients = juce::dsp::IIR::Coefficients<float>::makeBandPass(sampleRate, 1000.0f, 0.7f);

    highPassFilter.prepare(spec);
```

```
    highPassFilter.coefficients =
      juce::dsp::IIR::Coefficients<float>::makeHighPass(sampleRate, 20.0f);
}

// Function: releaseResources (Self-written)
// Purpose: Releases any resources acquired during the operation of the
  processor.
void AudioProcessorClass::releaseResources()
{
}

// Function: setLowPassFrequency (Self-written)
// Purpose: Sets the frequency of the low pass filter, with validation and
  error handling.
// Inputs:
//   frequency - The new frequency for the low pass filter.
void AudioProcessorClass::setLowPassFrequency(double frequency)
{
    if (frequency <= 0.0 || frequency > 20000.0)
    {
        std::cerr << "Invalid frequency value: " << frequency << ".
          Frequency should be in the range (0, 20000]" << std::endl;
        frequency = 1000.0;
    }

    auto coefficients = juce::dsp::IIR::Coefficients<float>::makeLowPass
      (currentSampleRate, frequency);
    lowPassFilter.coefficients = coefficients;
    DBG("LowPass Frequency changed to: " + juce::String(frequency));
}
// Function: setBandPassFrequency (Self-written)
// Purpose: Sets the frequency of the band pass filter, with validation and
  error handling.
// Inputs:
//   frequency - The new frequency for the band pass filter.
void AudioProcessorClass::setBandPassFrequency(double frequency)
{
    if (frequency <= 0.0 || frequency > 20000.0)
    {
        // Log the error
        std::cerr << "Invalid frequency value: " << frequency << ".
          Frequency should be in the range (0, 20000]" << std::endl;

        // Set a fallback value
        frequency = 1000.0;  // Using 1000 Hz as the fallback value. You can
           choose any other valid value.
    }

    auto coefficients = juce::dsp::IIR::Coefficients<float>::makeBandPass
      (currentSampleRate, frequency, 0.7f);
    bandPassFilter.coefficients = coefficients;
}
// Function: setHighPassFrequency (Self-written)
```

```cpp
// Purpose: Sets the frequency of the high pass filter, with validation and error handling.
// Inputs:
//   frequency - The new frequency for the band pass filter.
void AudioProcessorClass::setHighPassFrequency(double frequency)
{
    if (frequency <= 0.0 || frequency > 20000.0)
    {
        // Log the error
        std::cerr << "Invalid frequency value: " << frequency << ". Frequency should be in the range (0, 20000]" << std::endl;

        // Set a fallback value
        frequency = 500.0;  // Using 500 Hz as the fallback value. You can choose any other valid value.
    }

    auto coefficients = juce::dsp::IIR::Coefficients<float>::makeHighPass(currentSampleRate, frequency);
    highPassFilter.coefficients = coefficients;

}


// This method processes an audio block which consists of several audio channels
void AudioProcessorClass::processAudioBlock(juce::AudioBuffer<float>& buffer)
{
    // Check if the buffer has any channels or samples, if not log an error and return
    if (buffer.getNumChannels() == 0 || buffer.getNumSamples() == 0)
    {
        DBG("Error: Buffer has no channels or no samples");
        return;
    }

    // Loop over all channels in the buffer and process each one individually
    for (int channel = 0; channel < buffer.getNumChannels(); ++channel)
    {
        // Calling the function to process a single channel with the write pointer to the channel data and the number of samples
        processSingleChannel(buffer.getWritePointer(channel), buffer.getNumSamples());
    }
}

// This method processes a single channel of audio data
void AudioProcessorClass::processSingleChannel(float* channelData, int numSamples)
{
    // Creating an audio block object for a single channel by passing the
```

```
      channel data pointer and the number of samples
    juce::dsp::AudioBlock<float> audioBlock(&channelData, 1, numSamples);

    // Creating a processing context using the created audio block which
      will replace the original data with the processed data
    juce::dsp::ProcessContextReplacing<float> context(audioBlock);

    // Applying various filters sequentially to the audio data in the
      context
    lowPassFilter.process(context); // Applying low pass filter
    bandPassFilter.process(context); // Applying band pass filter
    highPassFilter.process(context); // Applying high pass filter
}
```

```
/*
==============================================================================
    AudioProcessorClass.h
    Created: 10 Sep 2023 12:29:54pm
    Author:  Ali
==============================================================================
*/

#pragma once

#include <JuceHeader.h>

// This class manages the audio processing in your application, including filtering operations.
class AudioProcessorClass
{
public:
    // Constructor: Initializes member variables to default values (Self-written)
    AudioProcessorClass();

    // Destructor: Cleans up the resources before the object is deleted (Self-written)
    ~AudioProcessorClass();

    // Function: prepareToPlay (Self-written)
    // Purpose: Sets up the audio processor to be ready for play.
    // Inputs:
    // - double sampleRate: The sample rate to be used for audio processing.
    // - int samplesPerBlock: The number of samples in each block of audio data.
    void prepareToPlay(double sampleRate, int samplesPerBlock);

    // Function: releaseResources (Self-written)
    // Purpose: Releases any resources that were in use by the processor.
    void releaseResources();

    // Function: setLowPassFrequency (Self-written)
    // Purpose: Sets the frequency for the low pass filter and updates its coefficients accordingly.
    // Inputs:
    // - double frequency: The new frequency value for the low pass filter.
    void setLowPassFrequency(double frequency);

    // Function: setBandPassFrequency (Self-written)
    // Purpose: Sets the frequency for the band pass filter and updates its coefficients accordingly.
    // Inputs:
    // - double frequency: The new frequency value for the band pass filter.
```

```
    void setBandPassFrequency(double frequency);

    // Function: setHighPassFrequency (Self-written)
    // Purpose: Sets the frequency for the high pass filter and updates its
      coefficients accordingly.
    // Inputs:
    // - double frequency: The new frequency value for the high pass filter.
    void setHighPassFrequency(double frequency);

    // Function: processAudioBlock (Self-written)
    // Purpose: Processes an audio buffer, applying filters to the audio
      data.
    // Inputs:
    // - juce::AudioBuffer<float>& buffer: A reference to the buffer
      containing the audio data to be processed.
    void processAudioBlock(juce::AudioBuffer<float>& buffer);

    // Function: processSingleChannel (Self-written)
    // Purpose: Processes a single channel of audio data.
    // Inputs:
    // - float* channelData: A pointer to the data for the channel to be
      processed.
    // - int numSamples: The number of samples in the channel data.
    void processSingleChannel(float* channelData, int numSamples);

private:
    // Filters and state variables for the audio processor (Self-written)
    juce::dsp::IIR::Filter<float> lowPassFilter;
    juce::dsp::IIR::Filter<float> bandPassFilter;
    juce::dsp::IIR::Filter<float> highPassFilter;

    double lowPassFrequency = 2000.0; // Initial value for low pass filter
      frequency
    double bandPassFrequency = 1000.0; // Initial value for band pass filter
       frequency
    double highPassFrequency = 500.0; // Initial value for high pass filter
      frequency

    double currentSampleRate;
    double lastSampleRate;
    int lastSamplesPerBlock;
};
```

```cpp
/*
==============================================================================
    AudioProcessorClass.h
    Created: 10 Sep 2023 12:29:54pm
    Author:  Ali
==============================================================================
*/

#include <JuceHeader.h>
#include "CoordinatePlot.h"
#include <iomanip>
#include <sstream>

//==============================================================================
CoordinatePlot::CoordinatePlot()
{
    // The constructor initializes grid settings, coordinate values, and child components
    // No inputs.
    // Outputs are internal settings necessary for the coordinate plot display and interaction.

    // Initialize grid line count, range and coordinates with default values
    setGridLineCount(); // Self-generated code
    setRange(); // Self-generated code
    initCoords(75.0f, 75.0f); // Self-generated code

    // Adding and making labels visible, and setting their justification type
    addAndMakeVisible(topLabel); // Self-generated code
    addAndMakeVisible(bottomLabel); // Self-generated code
    topLabel.setJustificationType(juce::Justification::centredTop); // Self-generated code
    bottomLabel.setJustificationType(juce::Justification::centredBottom); // Self-generated code
}

CoordinatePlot::~CoordinatePlot() {}
CoordinatePlot::Listener::Listener() {}
CoordinatePlot::Listener::~Listener() {}

void CoordinatePlot::paint(juce::Graphics& g)
{
    // This function paints the background, the plot, and the markers on the coordinate plot
    // Input: juce::Graphics& g, a reference to the graphics context that should be used to paint the component.
```

```
    // Output: The function modifies the graphics context to paint the
      necessary elements on the component.

    g.fillAll(getLookAndFeel().findColour
      (juce::ResizableWindow::backgroundColourId)); // Self-generated code

    // Drawing plot and marker with respective colours
    g.setColour(juce::Colours::grey);
    drawPlot(g); // Self-generated code

    g.setColour(juce::Colours::orange);
    drawMarker(g); // Self-generated code

    g.setColour(juce::Colours::white);
    if (markerMoved) { drawText(g); } // Self-generated code

    // Capture the raw range for resizing reference
    setRangeRaw(); // Self-generated code
}

void CoordinatePlot::resized()
{
    // This function updates settings and coordinates when the component is
      resized
    // No inputs.
    // Output: Updates internal settings necessary for adjusting the display
       to the new size.

    setSettings(); // Self-generated code
    updateCoords(); // Self-generated code

    // Set bounds for top and bottom labels
    topLabel.setBounds(getLocalBounds()); // Self-generated code
    bottomLabel.setBounds(getLocalBounds()); // Self-generated code
}

// This method is called when the mouse is pressed down on the component.
// Inputs:
//   - event: a reference to a juce::MouseEvent object representing the
  details of the mouse event
void CoordinatePlot::mouseDown(const juce::MouseEvent& event)
{
    // [Self-Generated] Debug statement to log the coordinates where the
      mouse was clicked. The getX() and getY() methods are used here but
      might need to be replaced with direct access to event.getMouseDownX()
      and event.getMouseDownY() to get the accurate clicked coordinates.
    DBG("Mouse Clicked on plot at: " << getX() << "," << getY());

    // [Self-Generated] Setting a flag indicating that the marker has moved
    markerMoved = true;

    // Changing the mouse cursor to NoCursor when mouse is pressed down
    setMouseCursor(juce::MouseCursor::NoCursor);
```

```cpp
    // [Self-Generated] Setting the coordinates of the marker to the point where the mouse was clicked
    setCoords(float(event.getMouseDownX()), float(event.getMouseDownY()));

    // [Self-Generated] Invoking interaction with the components listening to this plot
    interactWithComponent();

    // Requesting a repaint of the component to reflect the new marker position
    repaint();
}

// This method is called when the mouse is dragged on the component.
// Inputs:
//   - event: a reference to a juce::MouseEvent object representing the details of the mouse event
void CoordinatePlot::mouseDrag(const juce::MouseEvent& event)
{
    // [Self-Generated] Debug statement to log the current coordinates of the drag event
    DBG("Mouse dragged to: " << getX() << ", " << getY());

    // Retrieving the current position of the mouse
    juce::Point<int> rawPos(event.getPosition());
    float rawX = float(rawPos.getX());
    float rawY = float(rawPos.getY());

    // [Self-Generated] Updating the coordinates of the marker as the mouse is dragged
    setCoords(rawX, rawY);

    // [Self-Generated] Invoking interaction with the components listening to this plot
    interactWithComponent();

    // Requesting a repaint of the component to reflect the new marker position
    repaint();
}

// This method is called when the mouse button is released on the component.
// Inputs:
//   - event: a reference to a juce::MouseEvent object representing the details of the mouse event
void CoordinatePlot::mouseUp(const juce::MouseEvent& event)
{
    // Restoring the mouse cursor to its normal state when the mouse button is released
    setMouseCursor(juce::MouseCursor::NormalCursor);
}
```

```cpp
// [Self-Generated] This method notifies all the listeners about the change in value of the plot
void CoordinatePlot::interactWithComponent()
{
    // Invoking the coordPlotValueChanged method on all the listeners with this plot as the argument
    listeners.call([this](Listener& l) { l.coordPlotValueChanged(this); });
}

// [Self-Generated] Method to add a listener to this plot
// Inputs:
//   - l: pointer to the listener object to be added
void CoordinatePlot::addListener(Listener* l)
{
    listeners.add(l);
}

// [Self-Generated] Method to remove a listener from this plot
// Inputs:
//   - l: pointer to the listener object to be removed
void CoordinatePlot::removeListener(Listener* l)
{
    listeners.remove(l);
}

// [Self-Generated] Method to get the constrained X coordinate of the marker
// Outputs:
//   - returns the constrained X coordinate of the marker
float CoordinatePlot::getX()
{
    return constrain(coordsRaw['x']);
}

// [Self-Generated] Method to get the inverted and constrained Y coordinate of the marker
// Outputs:
//   - returns the inverted and constrained Y coordinate of the marker
float CoordinatePlot::getY()
{
    return invertCoord(constrain(coordsRaw['y']), range['min'], range['max']);
}

// [Self-Generated] Method to set the count of grid lines on the plot, ensuring it is even
// Inputs:
//   - lineCount: the desired number of grid lines
void CoordinatePlot::setGridLineCount(int lineCount)
{
    // Ensuring that the lineCount is even by decrementing it if it is odd
    if (lineCount % 2 == 1) { --lineCount; }
    gridLineCount = lineCount;
}
```

```cpp
// This function sets the range of values that can be displayed on the plot.
// [Self-Generated]
// Inputs:
//   - min: the minimum value of the range
//   - max: the maximum value of the range
void CoordinatePlot::setRange(float min, float max)
{
    range['min'] = min;
    range['max'] = max;
}

// This function initializes the coordinates of the plot to the given values.
// [Self-Generated]
// Inputs:
//   - rawX: the initial X coordinate
//   - rawY: the initial Y coordinate
void CoordinatePlot::initCoords(float rawX, float rawY)
{
    coordsRaw['x'] = rawX;
    coordsRaw['y'] = rawY;
}

// This function sets the coordinates if they are within the specified range.
// [Self-Generated]
// Inputs:
//   - rawX: the X coordinate to be set
//   - rawY: the Y coordinate to be set
void CoordinatePlot::setCoords(float rawX, float rawY)
{
    if (inRangeRaw(rawX, rawY)) { coordsRaw['x'] = rawX, coordsRaw['y'] = rawY; }
}

// This function updates the coordinates based on the initial range and the current size of the component.
// [Self-Generated]
void CoordinatePlot::updateCoords()
{
    // Computing the ratios based on the initial range
    double xRatio = double(coordsRaw['x'] / (rangeRaw['max'] - rangeRaw['min']));
    double yRatio = double(coordsRaw['y'] / (rangeRaw['max'] - rangeRaw['min']));

    // Calculating the new coordinates based on the current size and previous ratios
    float newX = float(right * xRatio);
    float newY = float(bottom * yRatio);

    setCoords(newX, newY);
```

```
}

// This function sets the raw range based on the local bounds of the
  component.
// [Self-Generated]
void CoordinatePlot::setRangeRaw()
{
    rangeRaw['min'] = getLocalBounds().getX();
    rangeRaw['max'] = getLocalBounds().getWidth();
}

// This function draws the plot, including an outline, axis, and grid.
// [Self-Generated]
// Inputs:
//   - g: a reference to a juce::Graphics object used for drawing
void CoordinatePlot::drawPlot(juce::Graphics& g)
{
    g.drawRect(getLocalBounds(), 3);// draw an outline around the component
    drawAxis(g);
    drawGrid(g);
}

// This function draws the x and y axis on the plot.
// [Self-Generated]
// Inputs:
//   - g: a reference to a juce::Graphics object used for drawing
void CoordinatePlot::drawAxis(juce::Graphics& g)
{
    // Drawing the x and y axis
    g.drawLine(left, midY, right, midY, 2);
    g.drawLine(midX, left, midX, bottom, 2);
}

// This function draws a grid on the plot.
// [Self-Generated]
// Inputs:
//   - g: a reference to a juce::Graphics object used for drawing
void CoordinatePlot::drawGrid(juce::Graphics& g)
{
    const float myDashLength[] = { 3, 3 };
    float offset = float(getLocalBounds().getWidth() / (gridLineCount + 2));

    for (int i = 0; i < (gridLineCount/2); ++i)
    {
        int d{ i + 1 }; //degrees away from axis
        //draw to left/right of Y-axis and top/bottom of X-axis
        g.drawDashedLine(juce::Line<float>(midX - offset * d, top, midX -
          offset * d, bottom),
            &myDashLength[0], 2, 1.0, 0);
        g.drawDashedLine(juce::Line<float>(midX + offset * d, top, midX +
          offset * d, bottom),
            &myDashLength[0], 2, 1.0, 0);
        g.drawDashedLine(juce::Line<float>(left, midY - offset * d, right,
```

```
          midY - offset * d),
            &myDashLength[0], 2, 1.0, 0);
        g.drawDashedLine(juce::Line<float>(left, midY + offset * d, right,
          midY + offset * d),
            &myDashLength[0], 2, 1.0, 0);
    }
}
// This function draws a marker on the plot.
// [Self-Generated]
// Inputs:
//   - g: a reference to a juce::Graphics object used for drawing
void CoordinatePlot::drawMarker(juce::Graphics& g)
{
    //set length of cursor
    float length = float(getLocalBounds().getWidth() / 15);

    //create lines
    juce::Line<float> lineH(juce::Point<float>(coordsRaw['x'] - length,
      coordsRaw['y']),
        juce::Point<float>(coordsRaw['x'] + length, coordsRaw['y']));
    juce::Line<float> lineV(juce::Point<float>(coordsRaw['x'], coordsRaw
      ['y'] - length),
        juce::Point<float>(coordsRaw['x'], coordsRaw['y'] + length));

    //draw lines
    g.drawLine(lineH, 2.0f);
    g.drawLine(lineV, 2.0f);
}

// This function draws the x and y coordinates as text on the plot.
// [Self-Generated]
// Inputs:
//   - g: a reference to a juce::Graphics object used for drawing
void CoordinatePlot::drawText(juce::Graphics& g)
{
    g.setFont(float(getWidth()/12));
    int textHeight = int(g.getCurrentFont().getHeight());

    //Draw Y
    std::stringstream streamY;
    streamY << std::fixed << std::setprecision(2) << getY();
    g.drawText(streamY.str(), int(midX), int(top), int(midX), textHeight,
        juce::Justification::centredLeft, true);

    //Draw X
    std::stringstream streamX;
    streamX << std::fixed << std::setprecision(2) << getX();
    g.drawText(streamX.str(), int(midX), int(midY), int(midX), textHeight,
        juce::Justification::centredRight, true);
}
// This function recalculates all the settings, updating the variables
  representing different points and boundaries in the local bounds of the
  component.
```

```cpp
// [Self-Generated]
void CoordinatePlot::setSettings()
{
    //recalculate all the settings
    midY = float(getLocalBounds().getCentreY());
    midX = float(getLocalBounds().getCentreX());
    left = float(getLocalBounds().getX());
    right = float(getLocalBounds().getRight());
    top = float(getLocalBounds().getY());
    bottom = float(getLocalBounds().getBottom());
}
// This function constrains a coordinate to be within a specific range,
  based on a transformation of ranges.
// [Self-Generated]
// Inputs:
//   - coord: the coordinate to be constrained
// Outputs:
//   - returns the constrained coordinate
float CoordinatePlot::constrain(float coord)
{
    float oldRangeMin = float(getLocalBounds().getX());
    float oldRangeMax = float(getLocalBounds().getWidth());
    float oldRange = oldRangeMax - oldRangeMin;
    float newRange = range['max'] - range['min'];

    float newValue = (((coord - oldRangeMin) * newRange) / oldRange) + range
      ['min'];
    return newValue;
}
// This function inverts a coordinate within a specified range.
// [Self-Generated]
// Inputs:
//   - coord: the coordinate to be inverted
//   - min: the minimum value of the range
//   - max: the maximum value of the range
// Outputs:
//   - returns the inverted coordinate
/**Inverts coord within a range between min and max*/
float CoordinatePlot::invertCoord(float coord, float min, float max)
{
    return (min + max) - coord;
}
// This function checks if the raw coordinates are within the valid range of
   the plot.
// [Self-Generated]
// Inputs:
//   - rawX: the raw X coordinate to be checked
//   - rawY: the raw Y coordinate to be checked
// Outputs:
//   - returns true if the coordinates are within the range, false otherwise
bool CoordinatePlot::inRangeRaw(float rawX, float rawY)
{
    return (rawX >= left && rawX <= right && rawY >= top && rawY <= bottom);
```

```cpp
}

void CoordinatePlot::setLabelText(const juce::String& topText, const juce::String& bottomText)
{
    // This function sets the text of the top and bottom labels in the coordinate plot.
    // Inputs:
    //   const juce::String& topText - the text to be set for the top label
    //   const juce::String& bottomText - the text to be set for the bottom label
    // Output: Modifies the topLabel and bottomLabel components to display the specified text.

    topLabel.setText(topText, juce::dontSendNotification); // Self-generated code
    bottomLabel.setText(bottomText, juce::dontSendNotification); // Self-generated code

    // Disabling the interception of mouse clicks on both labels
    topLabel.setInterceptsMouseClicks(false, false); // Self-generated code
    bottomLabel.setInterceptsMouseClicks(false, false); // Self-generated code
}
```

```cpp
/*
  ==============================================================================

    CoordinatePlot.h
    Created: 10 Aug 2023 5:23:41pm
    Author:  Ali

  ==============================================================================
*/

#pragma once

#include <JuceHeader.h>

//==============================================================================
/*
   A class to represent a coordinate plot component. This component allows for
   plotting and visualizing 2D coordinates with various customization options
   such as setting ranges, grid lines, and labels.

   This class inherits from juce::Component and juce::SettableTooltipClient.
*/
class CoordinatePlot : public juce::Component,
    public juce::SettableTooltipClient
{
public:
    // Default constructor
    CoordinatePlot();

    // Destructor
    ~CoordinatePlot() override;

    // Overrides paint method from juce::Component to draw the component
    void paint(juce::Graphics&) override;

    // Overrides resized method from juce::Component to update component size
    void resized() override;

    // Event handler for mouse down events
    void mouseDown(const juce::MouseEvent& event) override;

    // Event handler for mouse up events
    void mouseUp(const juce::MouseEvent& event) override;
```

```
    // Event handler for mouse drag events
    void mouseDrag(const juce::MouseEvent& event) override;

    /**
     * Set the number of grid lines shown on the Coordinate Plot.
     * @param lineCount: The number of grid lines to be displayed.
     *                   Defaults to 4 (4 vertical, 4 horizontal).
     *                   Reduces lineCount by 1 if odd. Uses default if
       lineCount < 2.
     * - Created without assistance.
     */
    void setGridLineCount(int lineCount = 4);

    /**
     * Set the range of values for the coordinate plot.
     * @param min: The minimum value of the range.
     * @param max: The maximum value of the range.
     * - Created without assistance.
     */
    void setRange(float min = 0.0f, float max = 1.0f);

    /**
     * Get the current x coordinate.
     * @return The current x coordinate.
     * - Created without assistance.
     */
    float getX();

    /**
     * Get the current y coordinate.
     * @return The current y coordinate.
     * - Created without assistance.
     */
    float getY();

    // Nested class to represent a listener for the coordinate plot
    class Listener
    {
    public:
        // Constructor
        Listener();

        // Destructor
        ~Listener();

        // Pure virtual function to define the callback for listener
        virtual void coordPlotValueChanged(CoordinatePlot* coordinatePlot) =
           0;
    };

    // Calls the callback method on all registered listeners
    void interactWithComponent();
```

```
    // Adds a listener to the list of registered listeners
    void addListener(Listener* l);

    // Removes a listener from the list of registered listeners
    void removeListener(Listener* l);

    // Sets the label texts for the top and bottom labels
    void setLabelText(const juce::String& topText, const juce::String&
      bottomText);

private:
    // A list of registered listeners
    juce::ListenerList<Listener> listeners;

    // A map to hold the raw coordinates
    std::map<char, float> coordsRaw;

    // A map to hold the raw range values
    std::map<char, float> rangeRaw;

    // Set the raw range values based on the current component bounds
    void setRangeRaw();

    // Initialize the coordinates with raw values
    void initCoords(float rawX, float rawY);

    // Set the coordinates with raw values, if within range
    void setCoords(float rawX, float rawY);

    // Update the coordinates based on the range and the component size
    void updateCoords();

    // Positional settings variables
    float midY, midX, left, right, top, bottom;

    // Function to set positional settings based on component bounds
    void setSettings();

    // User settings
    int gridLineCount; // The number of grid lines
    std::map<char, float> range; // A map to hold the range values

    // Drawing methods
    void drawPlot(juce::Graphics& g);
    void drawAxis(juce::Graphics& g);
    void drawGrid(juce::Graphics& g);
    void drawMarker(juce::Graphics& g);
    void drawText(juce::Graphics& g);

    // Indicates if the marker was moved
    bool markerMoved{ false };

    // Method to constrain coordinates within a range
```

```
    float constrain(float coord);

    // Method to invert coordinates within a given range
    float invertCoord(float coord, float min, float max);

    // Method to check if the raw coordinates are within range
    bool inRangeRaw(float rawX, float rawY);

    // Label components for displaying text at the top and bottom of the
      plot
    juce::Label topLabel;
    juce::Label bottomLabel;

    // Macro to declare non-copyable class and leak detector
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(CoordinatePlot)
};
```

```cpp
/*
==============================================================================

    CustomLookAndFeel.cpp
    Created: 10 Aug 2023 5:23:41pm
    Author:  Ali

==============================================================================
*/

#include "CustomLookAndFeel.h"

// Function to draw button background with custom colours for different states
// Inputs: Graphics context (g), Reference to the button, Background color, MouseOver and ButtonDown states
// Output: None (modifies the button appearance in the GUI)
void CustomLookAndFeel::drawButtonBackground(juce::Graphics& g, juce::Button& button,
    const juce::Colour& backgroundColour, bool isMouseOverButton, bool isButtonDown)
{
    // (Self-written code) Gets the bounds of the button area
    juce::Rectangle<int> buttonArea = button.getLocalBounds();
    juce::Colour buttonColor;

    // (Self-written code) Modifies the button color based on its state (clicked, hovered, or normal)
    if (isButtonDown)
    {
        buttonColor = juce::Colours::red;
    }
    else if (isMouseOverButton)
    {
        buttonColor = juce::Colours::midnightblue;
    }
    else
    {
        buttonColor = backgroundColour;
    }

    // Drawing the button with the designated color and a rounded rectangle shape
    g.setColour(buttonColor);
    g.fillRoundedRectangle(buttonArea.toFloat(), 10);

    // Drawing the border of the button
    g.setColour(juce::Colours::black);
    g.drawRoundedRectangle(buttonArea.toFloat(), 10, 1);
```

```
}

// Function to draw linear slider with a custom appearance
// Inputs: Graphics context (g), Slider properties (position, size, style)
  and reference to the slider object
// Output: None (modifies the slider appearance in the GUI)
void CustomLookAndFeel::drawLinearSlider(juce::Graphics& g, int x, int y,
  int width, int height,
    float sliderPos, float minSliderPos, float maxSliderPos,
    const juce::Slider::SliderStyle style, juce::Slider& slider)
{
    // (Self-written code) Setting the background color of the slider
    g.fillAll(juce::Colours::black);

    // Drawing the slider bar with a custom color and line thickness
    g.setColour(juce::Colour::fromRGB(0, 127, 255));
    g.drawLine(x, y + height / 2, x + width, y + height / 2, 2.0f);

    // (Self-written code) Drawing the thumb of the slider with a custom
      shape and color
    if (style == juce::Slider::LinearHorizontal || style ==
      juce::Slider::LinearVertical)
    {
        float thumbWidth = getSliderThumbRadius(slider);
        juce::Rectangle<float> thumbArea;

        // Setting the area of the thumb based on the slider style
        if (style == juce::Slider::LinearVertical)
            thumbArea = juce::Rectangle<float>(x + width * 0.5f - thumbWidth
               * 0.5f, sliderPos - thumbWidth, thumbWidth, thumbWidth *
              2.0f);
        else
            thumbArea = juce::Rectangle<float>(sliderPos - thumbWidth, y +
              height * 0.5f - thumbWidth * 0.5f, thumbWidth * 2.0f,
              thumbWidth);

        // Drawing the thumb as a rounded rectangle with a default shape but
           custom color
        g.setColour(slider.findColour(juce::Slider::thumbColourId));
        g.fillRoundedRectangle(thumbArea.reduced(1.0f), thumbWidth * 0.5f);
    }
}

// Function to draw button text with custom settings
// Inputs: Graphics context (g), Reference to the text button, MouseOver and
   ButtonDown states
// Output: None (modifies the button text appearance in the GUI)
void CustomLookAndFeel::drawButtonText(juce::Graphics& g, juce::TextButton&
  button, bool isMouseOverButton, bool isButtonDown)
{
    // (Self-written code) Getting the font based on the button height and
      text
    juce::Font font = getFontFromHeight(button.getHeight(),
```

```
      button.getButtonText());

    // Setting the colour and drawing the text in the button
    g.setColour(button.findColour
      (juce::TextButton::textColourOffId).withMultipliedAlpha
      (button.isEnabled() ? 1.0f : 0.5f));
    g.setFont(font);
    g.drawText(button.getButtonText(), button.getLocalBounds(),
      juce::Justification::centred, true);
}

// Function to derive font settings based on the button height and text
// Inputs: Height of the button and the text to be displayed on the button
// Output: Customized font object
juce::Font CustomLookAndFeel::getFontFromHeight(int height, const
  juce::String& text)
{
    // (Self-written code) Setting the font size based on the height and
      making it bold
    juce::Font font(height * 0.6f);
    font.setBold(true);
    return font;
}

// Function to draw waveform with custom aesthetics
// Inputs: Graphics context (g), Area to draw the waveform, Data array of
  the waveform, Size of the data array
// Output: None (draws the waveform in the GUI)
void CustomLookAndFeel::drawWaveform(juce::Graphics& g, const
  juce::Rectangle<int>& area, const float* data, int dataSize)
{
    if (dataSize <= 0) return;

    // (Self-written code) Setting up custom color and starting the path for
       waveform drawing
    g.setColour(juce::Colour::fromRGB(0, 128, 255));
    juce::Path waveform;
    waveform.startNewSubPath(area.getX(), juce::jmap(data[0], 0.0f, 1.0f,
      float(area.getBottom()), float(area.getY())));

    // Drawing the waveform with custom gradient and glow effect
    for (int i = 1; i < dataSize; ++i)
    {
        waveform.lineTo(area.getX() + i, juce::jmap(data[i], 0.0f, 1.0f,
          float(area.getBottom()), float(area.getY())));
    }

    // Applying a gradient fill and drawing the glow effect at the peaks of
      the waveform
    juce::ColourGradient gradient(juce::Colour::fromRGB(0, 128, 255), 0.0f,
      area.getY(),
        juce::Colour::fromRGB(0, 64, 128), 0.0f, area.getBottom(), false);
    g.setGradientFill(gradient);
```

```
    g.strokePath(waveform, juce::PathStrokeType(2.0f));
    g.setColour(juce::Colour::fromRGB(0, 128, 255).withAlpha(0.5f));
    g.fillPath(waveform);
}
```

```
/*
==============================================================================
    WaveformDisplay.h
    Created: 24 Jul 2023 10:55:44am
    Author:  Ali

==============================================================================
*/

#pragma once

#include <JuceHeader.h>


#include "CustomLookAndFeel.h"

//==============================================================================
/*
*/
class WaveformDisplay  : public juce::Component,
                         public juce::ChangeListener
{
public:
    WaveformDisplay(int _id,
                    juce::AudioFormatManager& formatManager,
                    juce::AudioThumbnailCache& thumbCache);
    ~WaveformDisplay() override;

    void paint (juce::Graphics&) override;
    void resized() override;
    void changeListenerCallback(juce::ChangeBroadcaster* source) override;
    void loadURL(juce::URL audioURL);
    /**set the relative position of the playhead*/
    void setPositionRelative(double pos);

    // Adding new members to handle mouse interaction and setting playback position
    void mouseDown(const juce::MouseEvent& event) override;
    void mouseDrag(const juce::MouseEvent& event) override;

    void  setWaveformData(const std::vector<float>& data);

    std::function<void(double)> onPositionChanged;

private:
    int id;
```

```
    bool fileLoaded;
    double position;
    juce::String fileName;
    juce::AudioThumbnail audioThumb;
    int dataSize = 0;
    std::vector<float> waveformData;
    CustomLookAndFeel customLookAndFeel;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (WaveformDisplay)
};
```