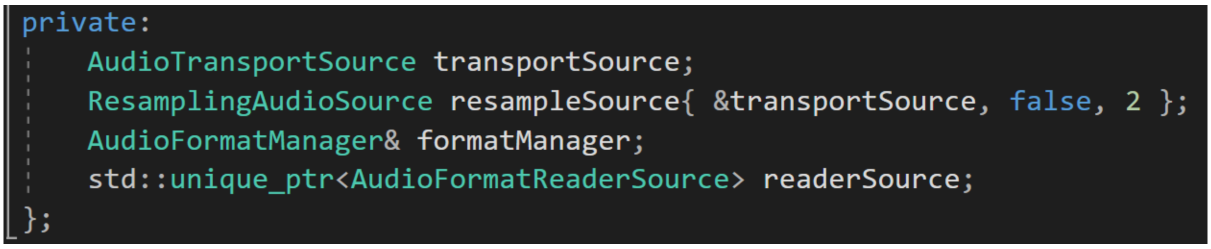
OBJECT ORIENTED PROGRAMMING
COURSEWORK FOR ENDTERM: OTODECKS
Student ID: 190488303

REPORT

R1: The application should contain all the basic functionality shown in class

R1A: Can load audio files into audio players

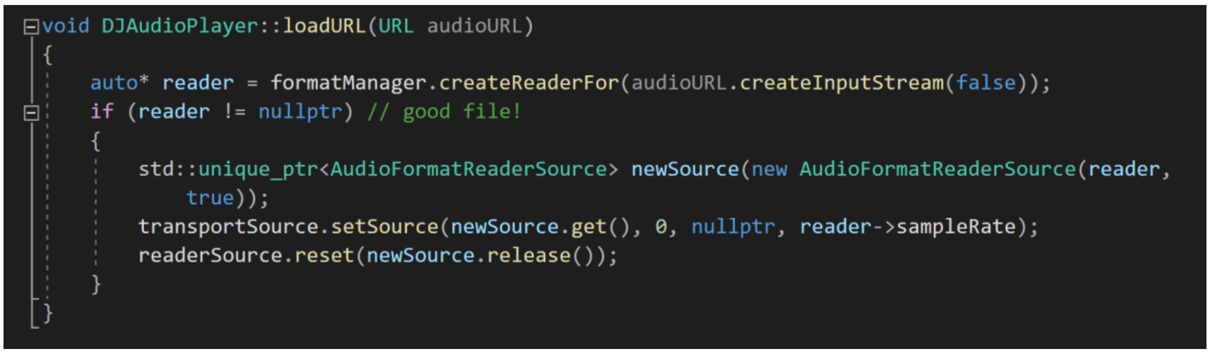


```
private:
    AudioTransportSource transportSource;
    ResamplingAudioSource resampleSource{ &transportSource, false, 2 };
    AudioFormatManager& formatManager;
    std::unique_ptr<AudioFormatReaderSource> readerSource;
};
```

In the class DJAudioPlayer, an AudioTransportSource object called transportSource is created while inheriting the juce library AudioSource. This allows us to set functions to an audio source, such as playing or stopping music tracks.

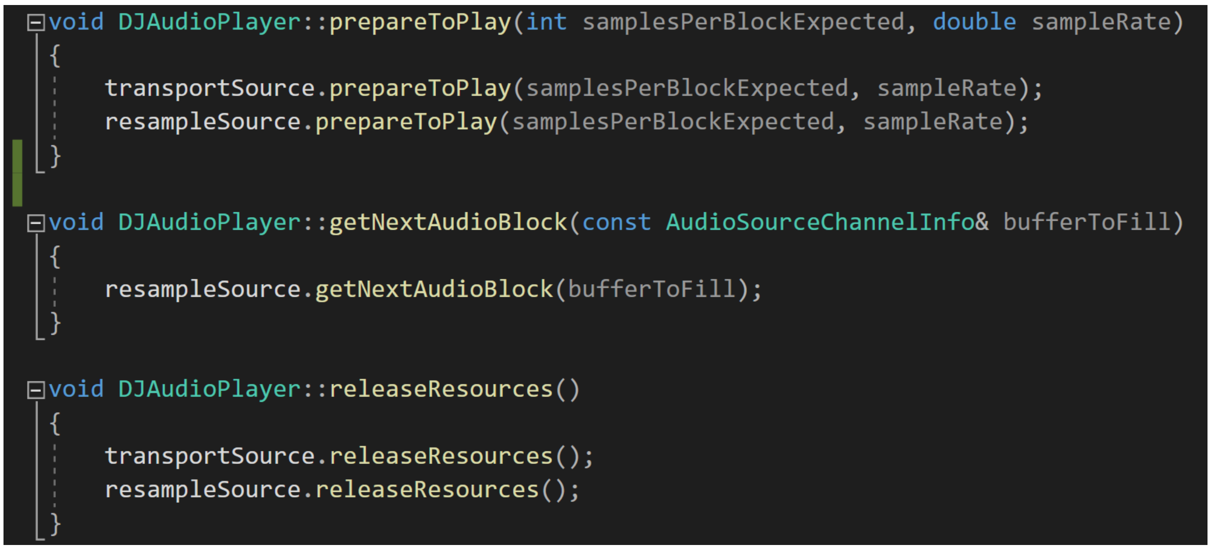
In the class DJAudioPlayer, a ResamplingAudioSource object called resampleSource is created while also inheriting from the juce library AudioSource. It takes in three parameters. Firstly, the transportSource which acts as an input source to read audio files from, secondly, a boolean value that is assigned false such that input source is not deleted when object is deleted, and thirdly, an integer value of 2 to represent the number of channels to process. This object functions to take an input source and change its sample rate.

Another object in the class DJAudioPlayer is AudioFormatManager, which inherits from juce library formatManager. It contains audio formats and decides which to use to open a audio file. The audioFormatReaderSource object called readerSource is then created and stored into a std::unique_ptr object to allow us to later form dynamic functions.



```
void DJAudioPlayer::loadURL(URL audioURL)
{
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
    if (reader != nullptr) // good file!
    {
        std::unique_ptr<AudioFormatReaderSource> newSource(new AudioFormatReaderSource(reader,
            true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset(newSource.release());
    }
}
```

In the DJAudioPlayer cpp file, the loadURL function is created to read and load files into the audio players. If the audio file can be read, it sets data from transportSource into parameters to be called later, and releases the data stored in readerSource.

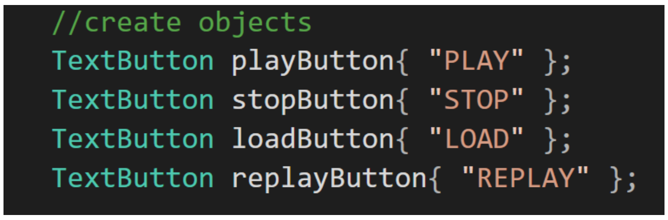


```cpp
void DJAudioPlayer::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{
    transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void DJAudioPlayer::getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
{
    resampleSource.getNextAudioBlock(bufferToFill);
}

void DJAudioPlayer::releaseResources()
{
    transportSource.releaseResources();
    resampleSource.releaseResources();
}
```

Additionally, the prepareToPlay() function puts transportSource() and resampleSource object in a prepared state, such that they can later be called in the function getNextAudioBlock to initialise any properties it might need. The releaseResources() function is then called to retrun the objects back into their unprepared state. All in all, these three functions are inherited from the AudioSource class in the juce library, and will work together to help load audio files into the audio players.

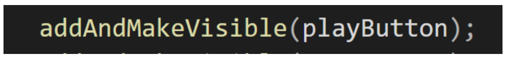
R1B: Can play two or more tracks

To play tracks, the DeckGUI class is implemented to create an interface and act as an audio player. There are two audio decks created which can play two different tracks, either separately or simultaneously.



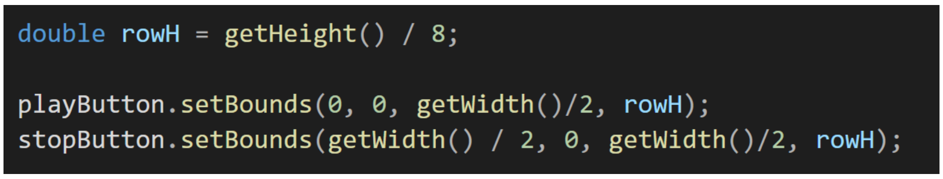
```
   //create objects
   TextButton playButton{ "PLAY" };
   TextButton stopButton{ "STOP" };
   TextButton loadButton{ "LOAD" };
   TextButton replayButton{ "REPLAY" };
```

The figure above shows how text button objects are created in the header file which in turn creates the play button.



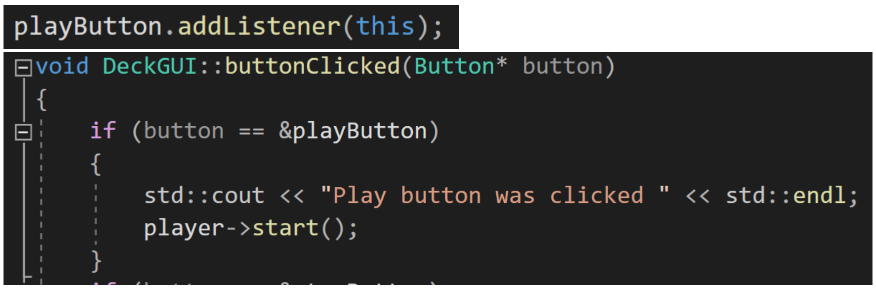
```
  addAndMakeVisible(playButton);
```

The following code above which can be found in the cpp file allows the play button created to be displayed visibly on the audio deck.



```
double rowH = getHeight() / 8;

playButton.setBounds(0, 0, getWidth()/2, rowH);
stopButton.setBounds(getWidth() / 2, 0, getWidth()/2, rowH);
```

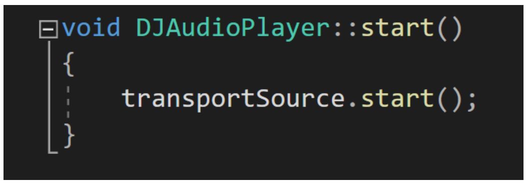
In the resized() function, setBounds is used to create a size and assign a location for the play button on the audio player deck.



```
playButton.addListener(this);
```

```
void DeckGUI::buttonClicked(Button* button)
{
    if (button == &playButton)
    {
        std::cout << "Play button was clicked " << std::endl;
        player->start();
    }
```

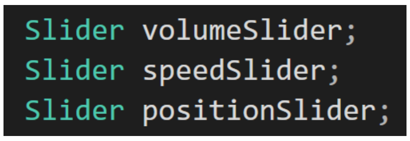
The above code adds a button listener to the playButton such that its ties the event of clicking on the button to the start() function in the DJAudioPlayer class, allowing it to play tracks.



```
void DJAudioPlayer::start()
{
    transportSource.start();
}
```

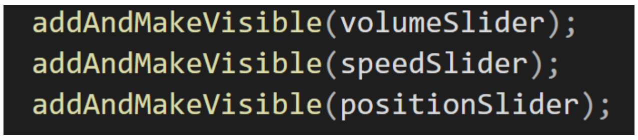
In the DJAudioPlayer class, the start() function as shown above works by allowing the object transportSource, which is the audio file, to start playing.

R1C: Can mix the tracks by varying each of their volume



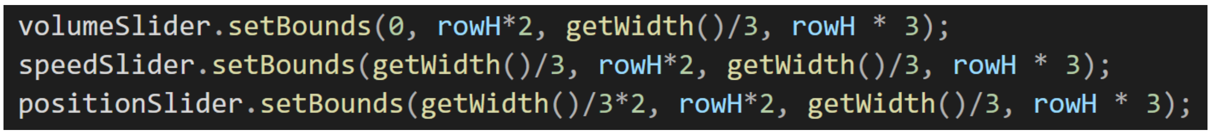
```
Slider volumeSlider;
Slider speedSlider;
Slider positionSlider;
```

In the DeckGUI header file, the object volumeSlider is created. It is inherited from the juce library using the slider class.



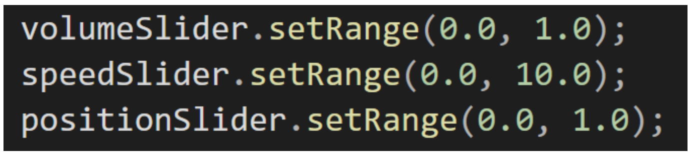
```
addAndMakeVisible(volumeSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(positionSlider);
```

In the DeckGUI cpp file, the addAndMakeVisible() function allows the volume slider created to be displayed visibly on the audio deck.



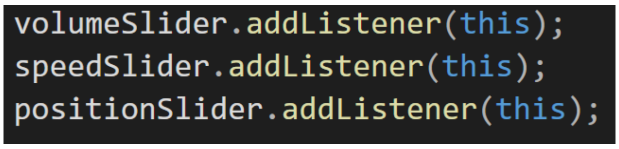
```
volumeSlider.setBounds(0, rowH*2, getWidth()/3, rowH * 3);
speedSlider.setBounds(getWidth()/3, rowH*2, getWidth()/3, rowH * 3);
positionSlider.setBounds(getWidth()/3*2, rowH*2, getWidth()/3, rowH * 3);
```

In the resized() function, setBounds is used to create a size and assign a location for the volume slider on the audio player deck.



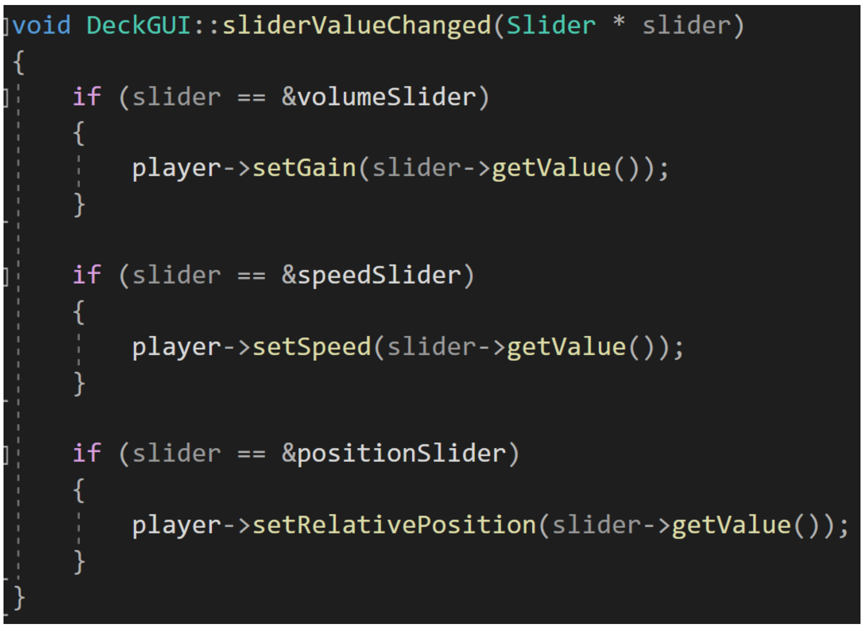
```
volumeSlider.setRange(0.0, 1.0);
speedSlider.setRange(0.0, 10.0);
positionSlider.setRange(0.0, 1.0);
```

The range of the volume slider is initialised in the DeckGUI class such that it ranges from 0.0, which is the minimum volume, to 1.0, which is maximum volume.

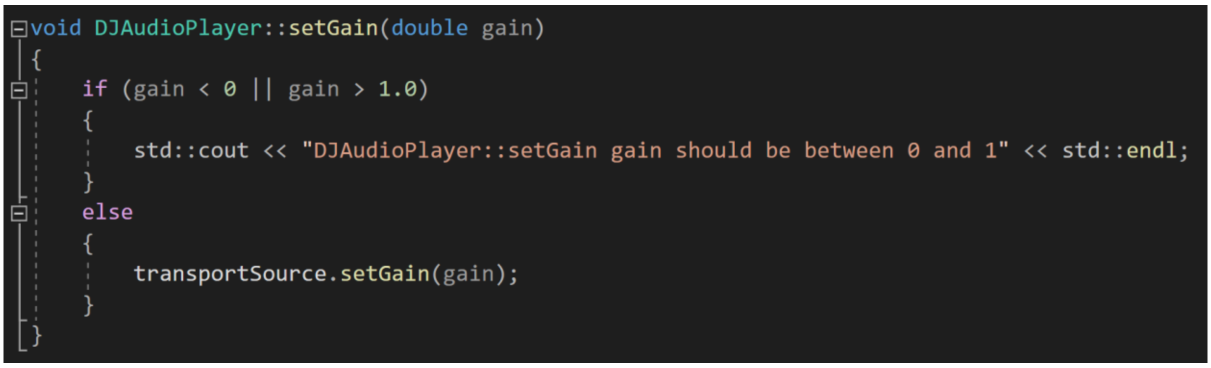


```
volumeSlider.addListener(this);
speedSlider.addListener(this);
positionSlider.addListener(this);
```

The above code adds a slider listener to the volume slider to link between an event where if the user changes the value of the slider, the volume of the track will be changed correspondingly.



```
void DeckGUI::sliderValueChanged(Slider * slider)
{
    if (slider == &volumeSlider)
    {
        player->setGain(slider->getValue());
    }

    if (slider == &speedSlider)
    {
        player->setSpeed(slider->getValue());
    }

    if (slider == &positionSlider)
    {
        player->setRelativePosition(slider->getValue());
    }
}
```

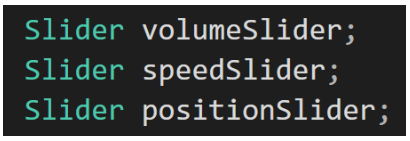


```
void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl;
    }
    else
    {
        transportSource.setGain(gain);
    }
}
```

This is further implemented in the first figure above within the function sliderValueChanged(), where the value of the slider is retrieved using getValue(). This value is then taken in by the setGain() function which we have created in the DJAudioPlayer class in the second figure above. The transportSource will then use the inherited setGain() function in the AudioSource to apply the corresponding volume changes to the output, which is the volume of the track.

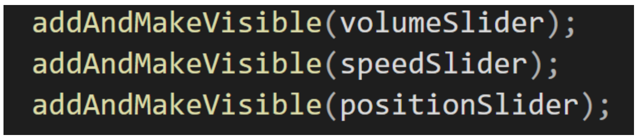
Since tracks can be loaded into both decks at the same time, one can then mix the tracks by playing both and varying each of their volumes respectively.

R1D: Can speed up and slow down the tracks



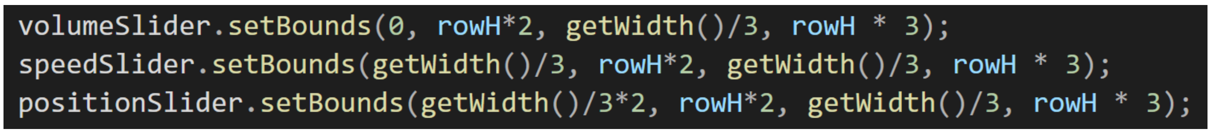
```
Slider volumeSlider;
Slider speedSlider;
Slider positionSlider;
```

In the DeckGUI header file, the object speedSlider is created. It is inherited from the juce library using the slider class.



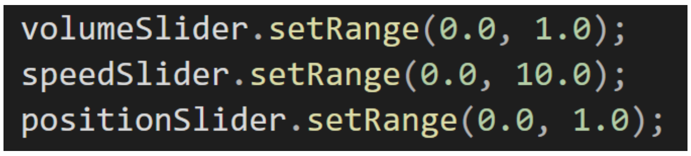
```
addAndMakeVisible(volumeSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(positionSlider);
```

In the DeckGUI cpp file, the addAndMakeVisible() function allows the speed slider created to be displayed visibly on the audio deck.



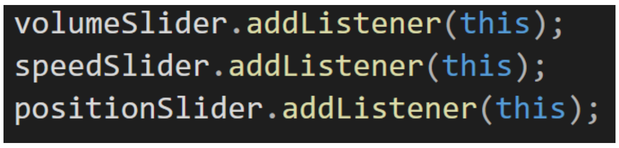
```
volumeSlider.setBounds(0, rowH*2, getWidth()/3, rowH * 3);
speedSlider.setBounds(getWidth()/3, rowH*2, getWidth()/3, rowH * 3);
positionSlider.setBounds(getWidth()/3*2, rowH*2, getWidth()/3, rowH * 3);
```

In the resized() function, setBounds is used to create a size and assign a location for the speed slider on the audio player deck.



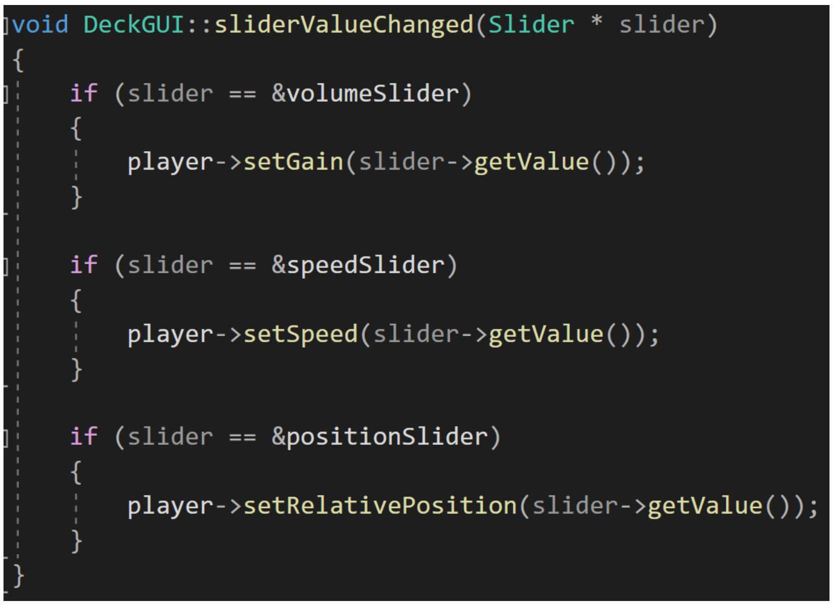
```
volumeSlider.setRange(0.0, 1.0);
speedSlider.setRange(0.0, 10.0);
positionSlider.setRange(0.0, 1.0);
```

The range of the speed slider is initialised in the DeckGUI class such that it ranges from 0.0, which is the lowest speed, to 10.0, which is highest speed.

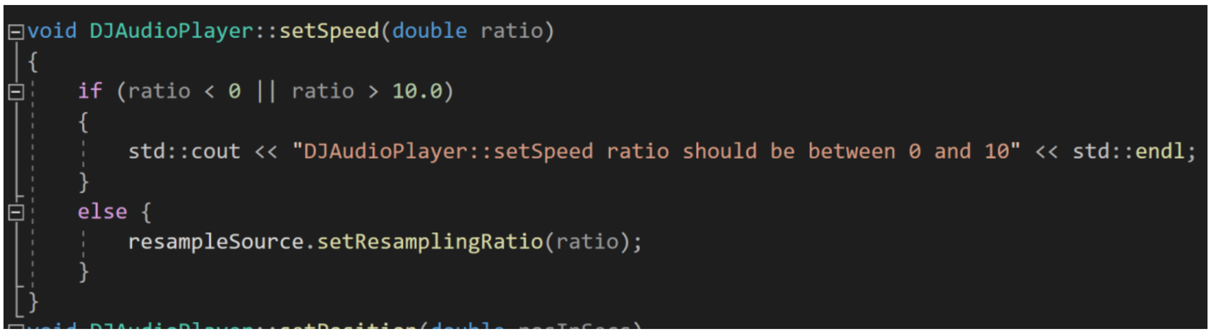


```
volumeSlider.addListener(this);
speedSlider.addListener(this);
positionSlider.addListener(this);
```

The above code adds a slider listener to the speed slider to link between an event where if the user changes the value of the slider, the speed of the track will be changed correspondingly.



```
void DeckGUI::sliderValueChanged(Slider * slider)
{
    if (slider == &volumeSlider)
    {
        player->setGain(slider->getValue());
    }

    if (slider == &speedSlider)
    {
        player->setSpeed(slider->getValue());
    }

    if (slider == &positionSlider)
    {
        player->setRelativePosition(slider->getValue());
    }
}
```



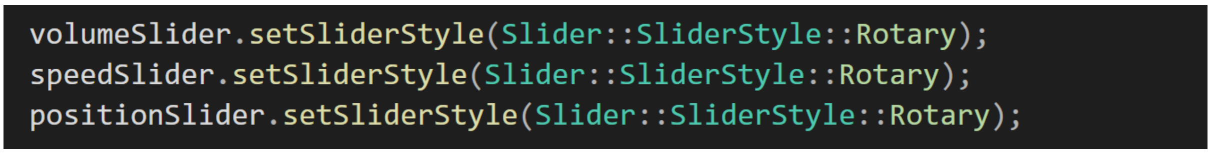
```
void DJAudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0 || ratio > 10.0)
    {
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 10" << std::endl;
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

This is further implemented in the first figure above within the function sliderValueChanged(), where the value of the slider is retrieved using getValue(). This value is then taken in by the setSpeed() function which we have created in the DJAudioPlayer class in the second figure above. The resampleSource will then use the inherited setResamplingRatio() function in the AudioSource to apply the corresponding speed change to the output, which is to either speed up or slow down the track.

R2: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/start.

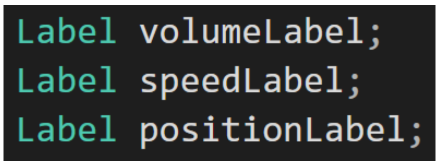
R2A: Component has custom graphics implemented in a paint function

The customization that I have chosen to implement for the deck control component is to change all three sliders for volume, speed, and position to rotary sliders, as well as attaching labels to the respective rotary sliders. The paint() function is used to render our component on the application interface.



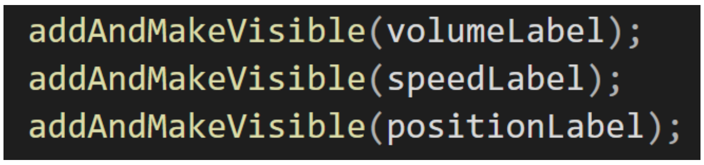
```
volumeSlider.setSliderStyle(Slider::SliderStyle::Rotary);
speedSlider.setSliderStyle(Slider::SliderStyle::Rotary);
positionSlider.setSliderStyle(Slider::SliderStyle::Rotary);
```

In the DeckGUI cpp file, the function setSliderStyle() is used to set the slider style of all three volume, speed and position slider objects to rotary sliders by inheriting the SliderStyle from the slider class. Hence, the volume, speed and position sliders are of rotary type.



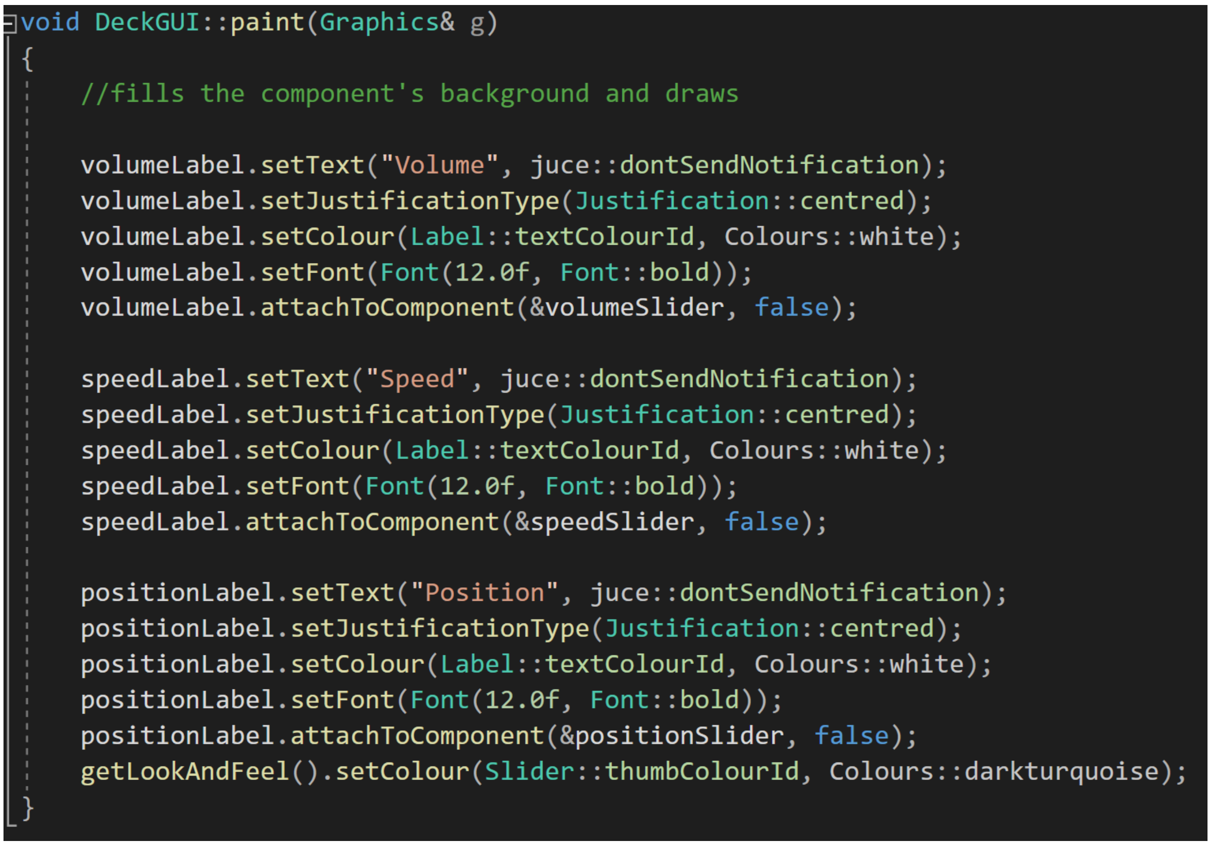
```
Label volumeLabel;
Label speedLabel;
Label positionLabel;
```

In the DeckGUI header file, the objects volumeLabel, speedLabel and positionLabel are initialised. They are inherited from the juce library using the label class.



```
addAndMakeVisible(volumeLabel);
addAndMakeVisible(speedLabel);
addAndMakeVisible(positionLabel);
```

In the DeckGUI cpp file, the addAndMakeVisible() function allows all three label objects, volumeLabel, speedLabel and positionLabel to be displayed visibly on the audio deck.

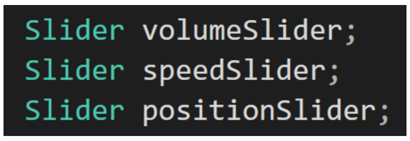


```
void DeckGUI::paint(Graphics& g)
{
    //fills the component's background and draws

    volumeLabel.setText("Volume", juce::dontSendNotification);
    volumeLabel.setJustificationType(Justification::centred);
    volumeLabel.setColour(Label::textColourId, Colours::white);
    volumeLabel.setFont(Font(12.0f, Font::bold));
    volumeLabel.attachToComponent(&volumeSlider, false);

    speedLabel.setText("Speed", juce::dontSendNotification);
    speedLabel.setJustificationType(Justification::centred);
    speedLabel.setColour(Label::textColourId, Colours::white);
    speedLabel.setFont(Font(12.0f, Font::bold));
    speedLabel.attachToComponent(&speedSlider, false);

    positionLabel.setText("Position", juce::dontSendNotification);
    positionLabel.setJustificationType(Justification::centred);
    positionLabel.setColour(Label::textColourId, Colours::white);
    positionLabel.setFont(Font(12.0f, Font::bold));
    positionLabel.attachToComponent(&positionSlider, false);
    getLookAndFeel().setColour(Slider::thumbColourId, Colours::darkturquoise);
}
```

The above snippet of code is taken from the paint() function in the DeckGUI class. For each label, I used the function setText() to take in a string parameter which will be displayed on the application as the name of each slider. They are named “Volume”, “Speed”, and “Position” for their respective slider with the same namesake. setText() also takes in another parameter which is set to not push any notification to users. setJustificationType() function is used to set the alignment of each label such that it is in the center of each slider. setColour() function takes in a textColourId integer and also paints all three labels white using the juce class colours. The setFont() function takes in font size and font type as parameter. The label sliders are set to have a font size of 12.0f, and are emboldened. Lastly, each slider object uses the attachToComponent() function to attach to their respective sliders, and takes in a false Boolean parameter to represent that they are attached to the top of their respective slider, position-wise.

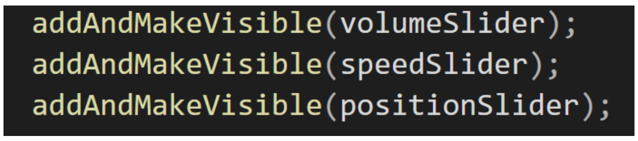
In the last line of code, the setColour() function is called such that the thumb color of all rotary sliders is set to dark turquoise.

R2B: Component enables the user to control the playback of a deck somehow.



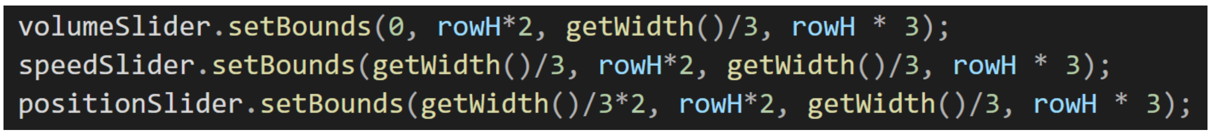
```
Slider volumeSlider;
Slider speedSlider;
Slider positionSlider;
```

In the DeckGUI header file, the object positionSlider is created. It is inherited from the juce library using the slider class.



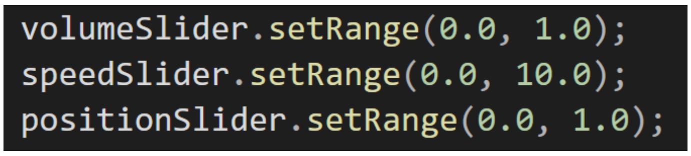
```
addAndMakeVisible(volumeSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(positionSlider);
```

In the DeckGUI cpp file, the addAndMakeVisible() function allows the position slider created to be displayed visibly on the audio deck.



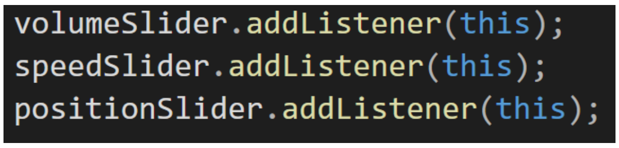
```
volumeSlider.setBounds(0, rowH*2, getWidth()/3, rowH * 3);
speedSlider.setBounds(getWidth()/3, rowH*2, getWidth()/3, rowH * 3);
positionSlider.setBounds(getWidth()/3*2, rowH*2, getWidth()/3, rowH * 3);
```

In the resized() function, setBounds is used to create a size and assign a location for the position slider on the audio player deck.



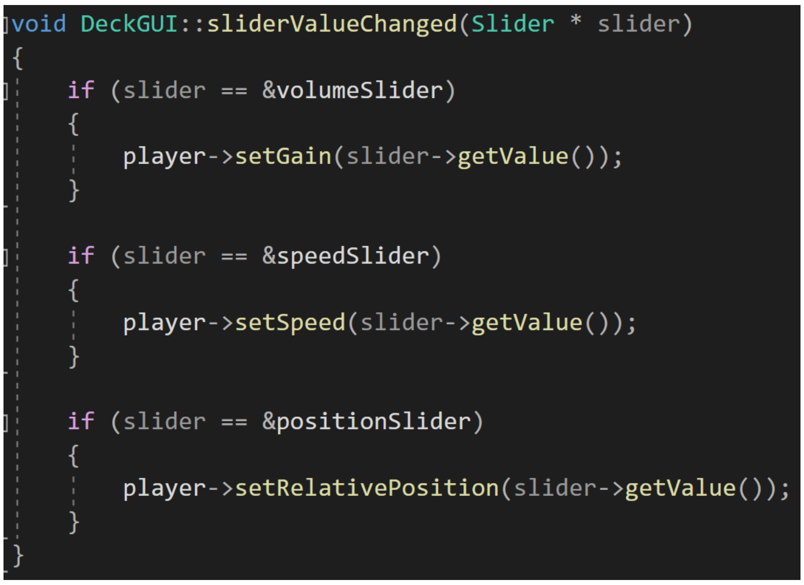
```
volumeSlider.setRange(0.0, 1.0);
speedSlider.setRange(0.0, 10.0);
positionSlider.setRange(0.0, 1.0);
```

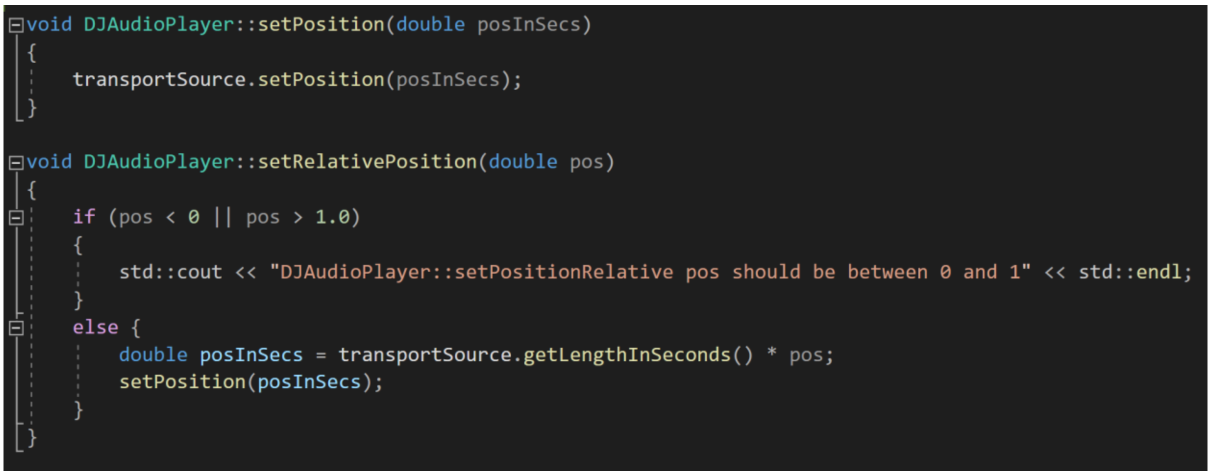
The range of the speed slider is initialised in the DeckGUI class such that it ranges from 0.0, which signifies the start of the track, to 1.0, which signifies the end of the track.



```
volumeSlider.addListener(this);
speedSlider.addListener(this);
positionSlider.addListener(this);
```

The above code adds a slider listener to the position slider to link between an event where if the user changes the value of the slider, the playback of the track will be changed correspondingly.

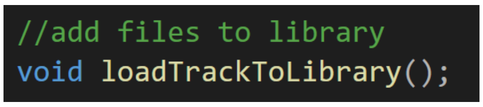

```
void DeckGUI::sliderValueChanged(Slider * slider)
{
    if (slider == &volumeSlider)
    {
        player->setGain(slider->getValue());
    }

    if (slider == &speedSlider)
    {
        player->setSpeed(slider->getValue());
    }

    if (slider == &positionSlider)
    {
        player->setRelativePosition(slider->getValue());
    }
}
```


```
void DJAudioPlayer::setPosition(double posInSecs)
{
    transportSource.setPosition(posInSecs);
}

void DJAudioPlayer::setRelativePosition(double pos)
{
    if (pos < 0 || pos > 1.0)
    {
        std::cout << "DJAudioPlayer::setPositionRelative pos should be between 0 and 1" << std::endl;
    }
    else {
        double posInSecs = transportSource.getLengthInSeconds() * pos;
        setPosition(posInSecs);
    }
}
```

This is further implemented in the first figure above within the function sliderValueChanged(), where the value of the slider is retrieved using getValue(). This value is then taken in by the setRelativePosition() function which we have created in the DJAudioPlayer class in the second figure above, and multiplied by the total number of seconds of the track being currently played, which is the transportSource, by using the function getLengthInSeconds(). It is stored in a new double variable posInSecs and used to call the setPosition() function, which sets the playback position to the corresponding timing in the track or transportSource. Hence, the control of a playback is implemented.

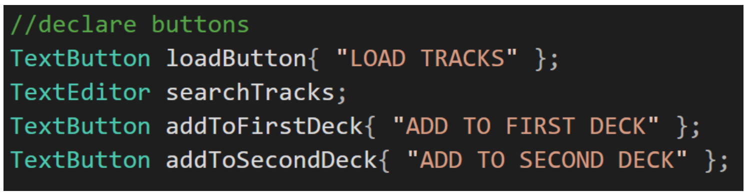
R3: Implementation of a music library component which allows the user to manage their music library

R3A: Component allows the user to add files to their library



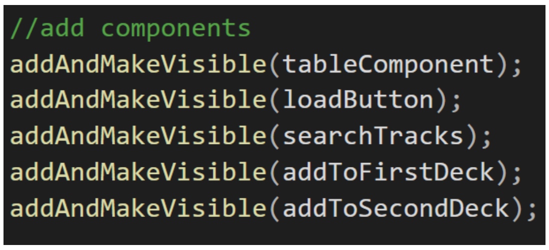
```
//add files to library
void loadTrackToLibrary();
```

Firstly, a function called loadTrackToLibrary() is implemented in the header file of the PlaylistComponent class.



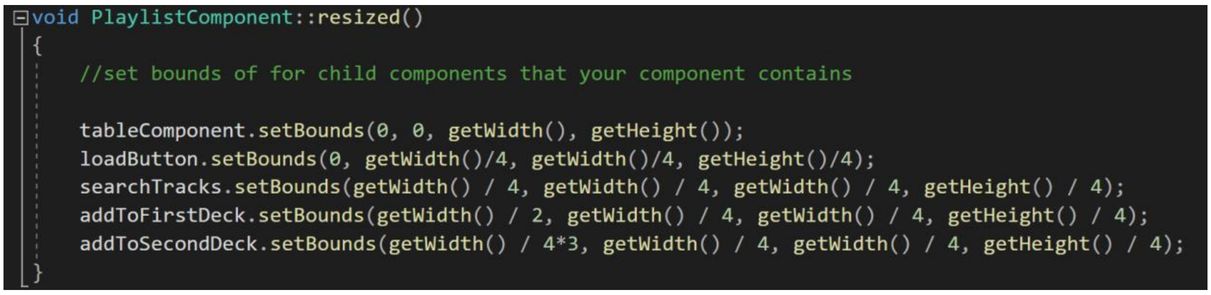
```
//declare buttons
TextButton loadButton{ "LOAD TRACKS" };
TextEditor searchTracks;
TextButton addToFirstDeck{ "ADD TO FIRST DECK" };
TextButton addToSecondDeck{ "ADD TO SECOND DECK" };
```

The figure above shows how text button objects are created in the header file which in turn creates the load button.



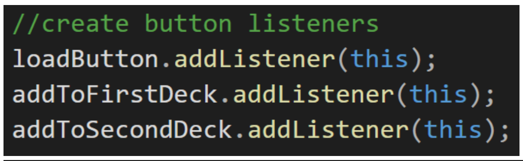
```
//add components
addAndMakeVisible(tableComponent);
addAndMakeVisible(loadButton);
addAndMakeVisible(searchTracks);
addAndMakeVisible(addToFirstDeck);
addAndMakeVisible(addToSecondDeck);
```

The following code above which can be found in the cpp file allows the load button created to be displayed visibly on the tableComponent.

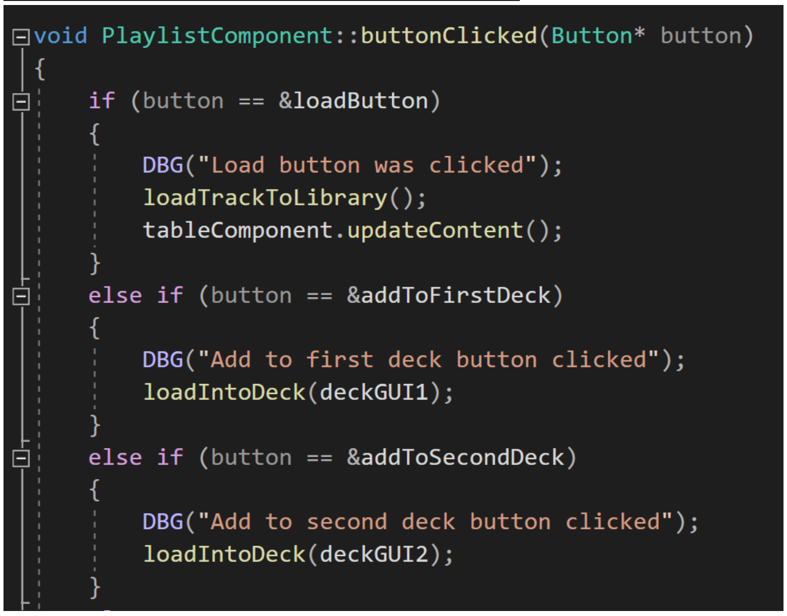


```
void PlaylistComponent::resized()
{
    //set bounds of for child components that your component contains

    tableComponent.setBounds(0, 0, getWidth(), getHeight());
    loadButton.setBounds(0, getWidth()/4, getWidth()/4, getHeight()/4);
    searchTracks.setBounds(getWidth() / 4, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToFirstDeck.setBounds(getWidth() / 2, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToSecondDeck.setBounds(getWidth() / 4*3, getWidth() / 4, getWidth() / 4, getHeight() / 4);
}
```

In the resized() function, setBounds is used to create a size and assign a location for the load button in the tableComponent.

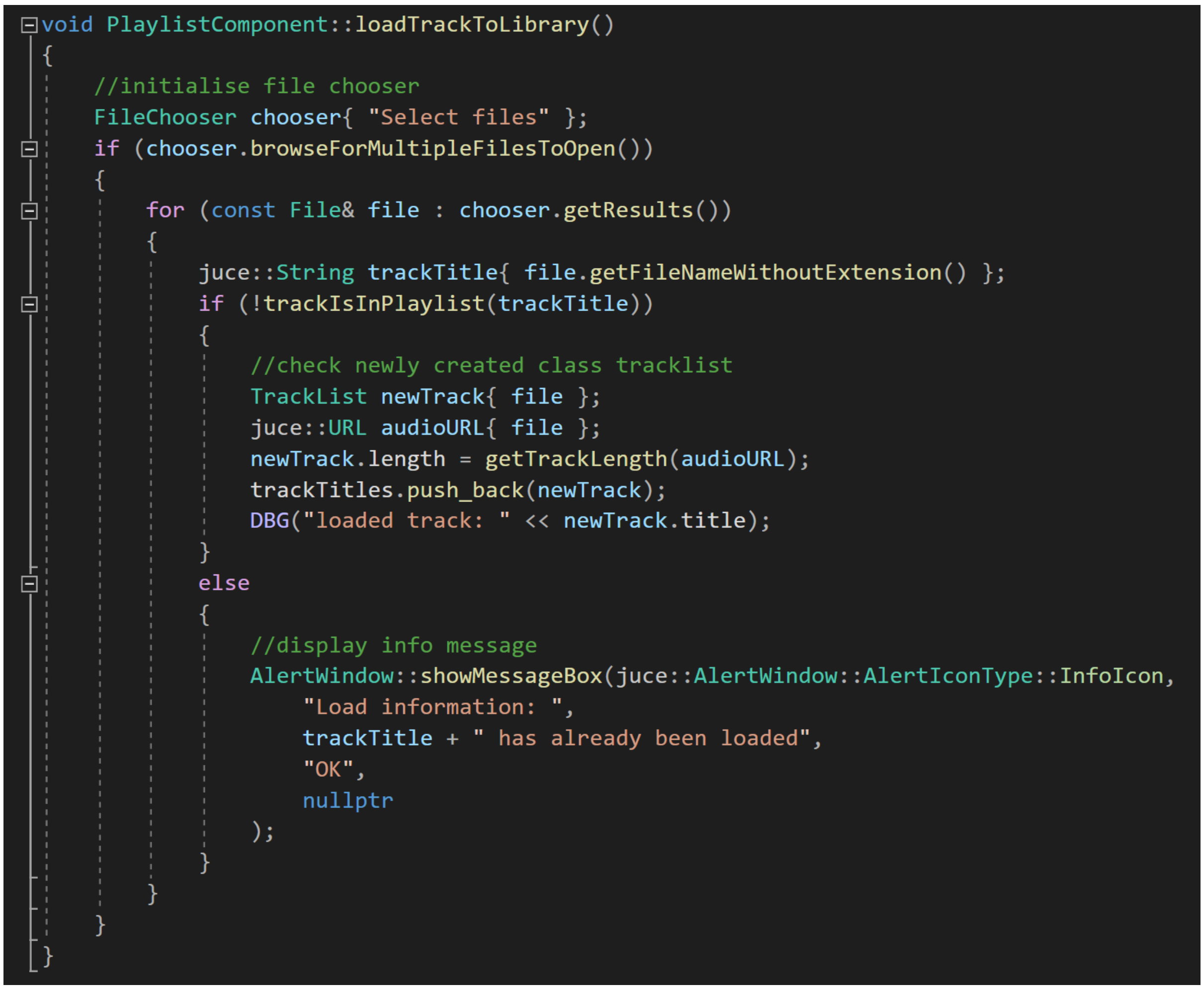


```cpp
//create button listeners
loadButton.addListener(this);
addToFirstDeck.addListener(this);
addToSecondDeck.addListener(this);
```

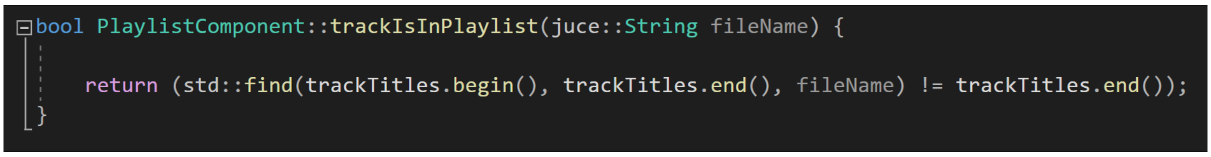


```cpp
void PlaylistComponent::buttonClicked(Button* button)
{
    if (button == &loadButton)
    {
        DBG("Load button was clicked");
        loadTrackToLibrary();
        tableComponent.updateContent();
    }
    else if (button == &addToFirstDeck)
    {
        DBG("Add to first deck button clicked");
        loadIntoDeck(deckGUI1);
    }
    else if (button == &addToSecondDeck)
    {
        DBG("Add to second deck button clicked");
        loadIntoDeck(deckGUI2);
    }
```

The above code adds a button listener to the loadButton such that its ties the event of clicking on the button to the loadTrackToLibrary() function, which will be explained later. The tableComponent will then be updated as new tracks are added to it.



```cpp
void PlaylistComponent::loadTrackToLibrary()
{
    //initialise file chooser
    FileChooser chooser{ "Select files" };
    if (chooser.browseForMultipleFilesToOpen())
    {
        for (const File& file : chooser.getResults())
        {
            juce::String trackTitle{ file.getFileNameWithoutExtension() };
            if (!trackIsInPlaylist(trackTitle))
            {
                //check newly created class tracklist
                TrackList newTrack{ file };
                juce::URL audioURL{ file };
                newTrack.length = getTrackLength(audioURL);
                trackTitles.push_back(newTrack);
                DBG("loaded track: " << newTrack.title);
            }
            else
            {
                //display info message
                AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
                    "Load information: ",
                    trackTitle + " has already been loaded",
                    "OK",
                    nullptr
                );
            }
        }
    }
}
```

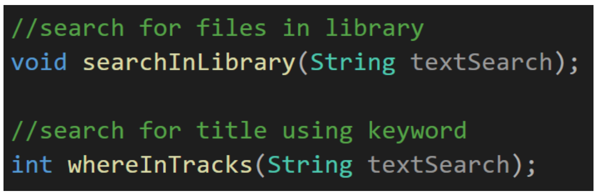


```cpp
bool PlaylistComponent::trackIsInPlaylist(juce::String fileName) {

    return (std::find(trackTitles.begin(), trackTitles.end(), fileName) != trackTitles.end());
}
```

In the PlayistComponent class, the loadTrackToLibrary() function is called. The FileChooser object is used to create a variable named chooser, which also displays a string when the file directory pops up. If one or more files are selected to be loaded into the tableComponent, browseForMultipleFilesToOpen() will be called such that for each chosen file, a string called trackTitle is declared to store name of the file without the file type, due to the function getFileNameWithoutExtension(). This is done by using the getResults() function which returns a list of all the files that were chosen. Afterwards, an if statement is used to check if the trackTitle added already exists in the tableComponent, by using the function trackIsInPlaylist() to search through the trackTitles for any repeated fileName. If there is no such track found, a new instance from the TrackList object called newTrack will be created to take in its URL and track length, by using the getTrackLength() function. It will then be pushed backinto the vector called trackTitles. However, if the track can already be found in the tableComponent, a message box will appear by using the AlertWindow class to create an AlertIconType called InfoIcon. Hence a message will be displayed to the user that the track has already been loaded.

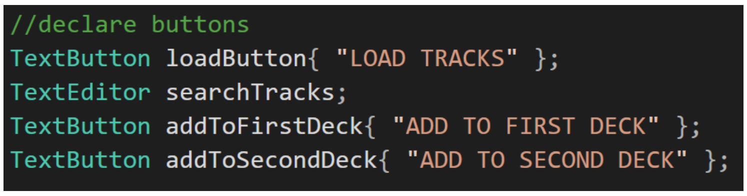
R3B: Component parses and displays meta data such as filename and song length

R3C: Component allows the user to search for files



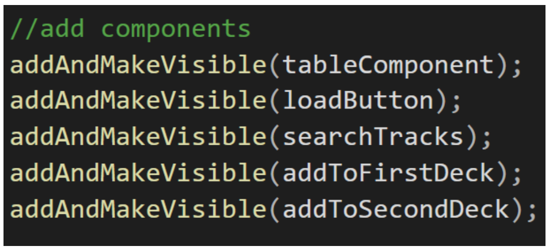
```
//search for files in library
void searchInLibrary(String textSearch);

//search for title using keyword
int whereInTracks(String textSearch);
```

A function called searchInLibrary() is implemented in the header file of the PlaylistComponent class.



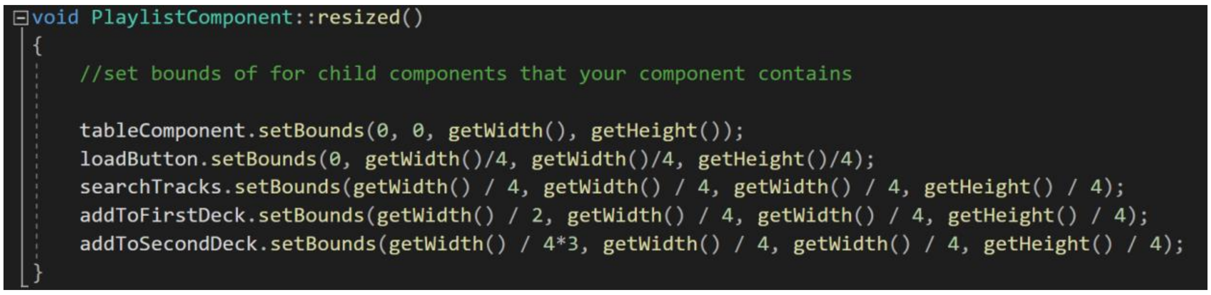
```
//declare buttons
TextButton loadButton{ "LOAD TRACKS" };
TextEditor searchTracks;
TextButton addToFirstDeck{ "ADD TO FIRST DECK" };
TextButton addToSecondDeck{ "ADD TO SECOND DECK" };
```

In the header file for the PlaylistComponent, a new variable called searchTracks is created from the TextEditor class from the juce library to form a search box to allow the user to search for tracks.



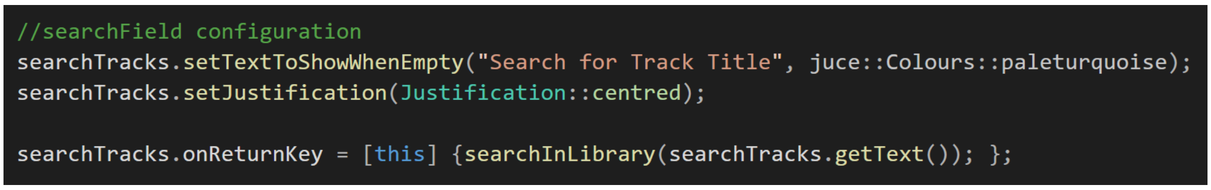
```
//add components
addAndMakeVisible(tableComponent);
addAndMakeVisible(loadButton);
addAndMakeVisible(searchTracks);
addAndMakeVisible(addToFirstDeck);
addAndMakeVisible(addToSecondDeck);
```

The following code above which can be found in the cpp file allows the searchTracks text editor that was previously created to be displayed visibly on the tableComponent.



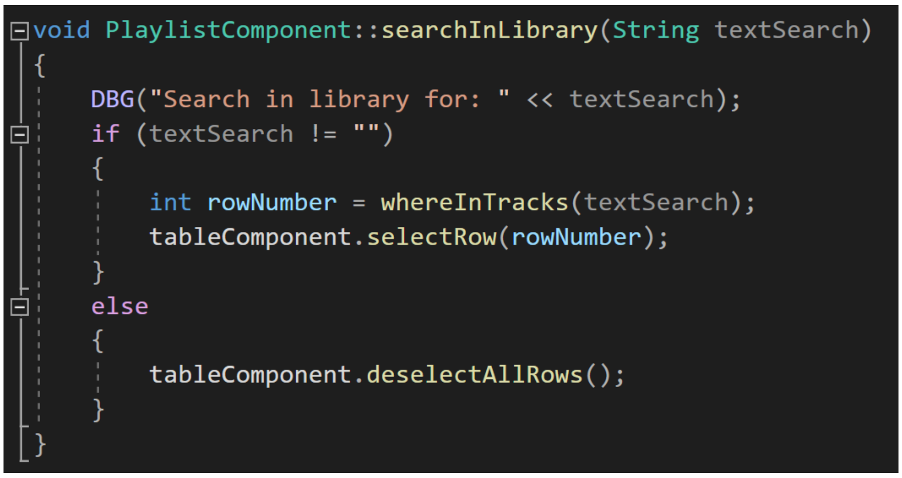
```
void PlaylistComponent::resized()
{
    //set bounds of for child components that your component contains

    tableComponent.setBounds(0, 0, getWidth(), getHeight());
    loadButton.setBounds(0, getWidth()/4, getWidth()/4, getHeight()/4);
    searchTracks.setBounds(getWidth() / 4, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToFirstDeck.setBounds(getWidth() / 2, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToSecondDeck.setBounds(getWidth() / 4*3, getWidth() / 4, getWidth() / 4, getHeight() / 4);
}
```

In the resized() function, setBounds is used to create a size and assign a location for the searchTracks text editor in the tableComponent.



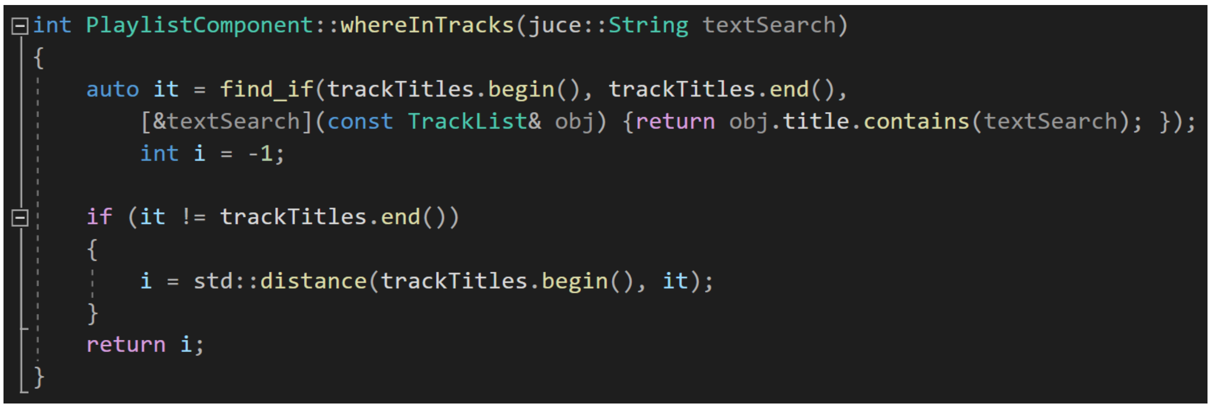
```
//searchField configuration
searchTracks.setTextToShowWhenEmpty("Search for Track Title", juce::Colours::paleturquoise);
searchTracks.setJustification(Justification::centred);

searchTracks.onReturnKey = [this] {searchInLibrary(searchTracks.getText()); };
```

The setTextToShowWhenEmpty() function allows the search box to display a tring message in the color pale turquoise, with a centred justification, by using the function setJustification(). The function onReturnKey() passes the user input into the search function searchInLibrary() when the user presses the return key. This helps to ensure that the text editor for the search function works by configuring the search field to respond to keywords .



```
void PlaylistComponent::searchInLibrary(String textSearch)
{
    DBG("Search in library for: " << textSearch);
    if (textSearch != "")
    {
        int rowNumber = whereInTracks(textSearch);
        tableComponent.selectRow(rowNumber);
    }
    else
    {
        tableComponent.deselectAllRows();
    }
}
```

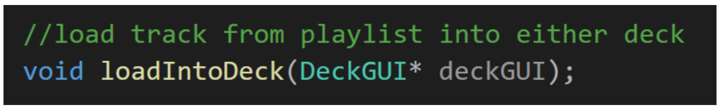
In the searchInLibrary() function, if there is user input, it will be passed into the function whereInTracks() and return the rowNumber of the corresponding track title. This will be further explained later. The tableComponent will then use the selectRow() function to select the row with that rowNumber to highlight to the user the results of the search. If it is not found, none of the rows will be highlighted as seen in the last line of code above.



```
int PlaylistComponent::whereInTracks(juce::String textSearch)
{
    auto it = find_if(trackTitles.begin(), trackTitles.end(),
        [&textSearch](const TrackList& obj) {return obj.title.contains(textSearch); });
        int i = -1;

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

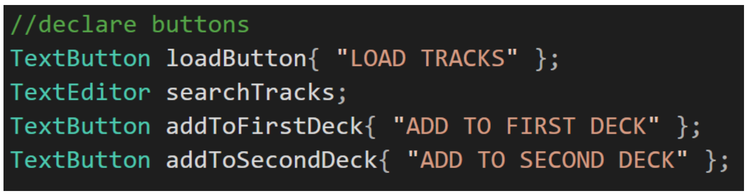
In the function whereInTracks(), all trackTitles from the object TrackList is searched by using the user input textSearch. If the trackTitle can be found, the distance of that track in the tableComponent from where the track title begins will be recorded in the integer i, and returned to searchInLibrary() as the rowNumber, where its purpose was already explained in the previous paragraph.

R3D: Component allows the user to load files from the library into a deck



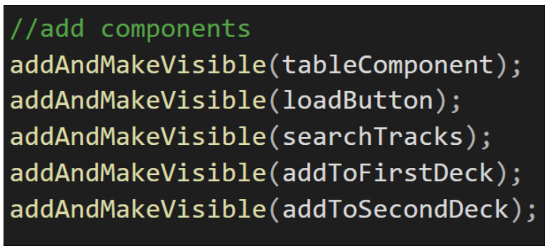
```
//load track from playlist into either deck
void loadIntoDeck(DeckGUI* deckGUI);
```

A function called loadIntoDeck() is implemented in the header file of the PlaylistComponent class.



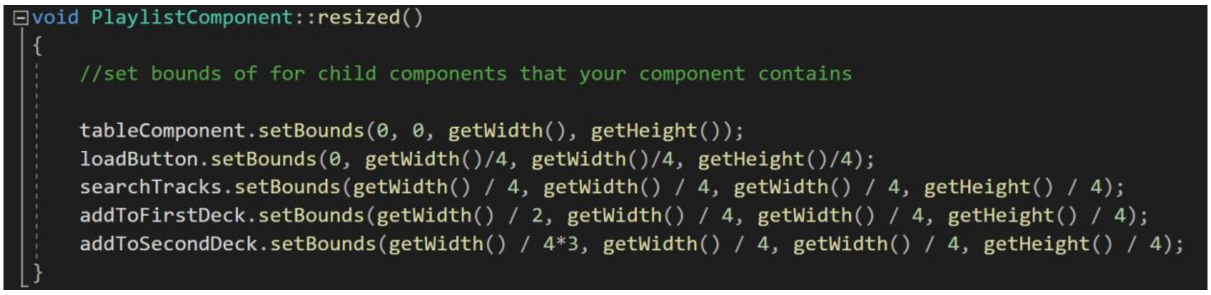
```
//declare buttons
TextButton loadButton{ "LOAD TRACKS" };
TextEditor searchTracks;
TextButton addToFirstDeck{ "ADD TO FIRST DECK" };
TextButton addToSecondDeck{ "ADD TO SECOND DECK" };
```

In the header file for the PlaylistComponent, two new variables called addToFirstDeck and addToSecondDeck are created by inheriting from the juce library in the TextButton class to form buttons to allow users to load files from playlistComponent into deck.



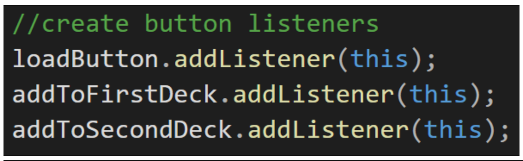
```
//add components
addAndMakeVisible(tableComponent);
addAndMakeVisible(loadButton);
addAndMakeVisible(searchTracks);
addAndMakeVisible(addToFirstDeck);
addAndMakeVisible(addToSecondDeck);
```

The following code above which can be found in the cpp file allows the addToFirstDeck and addToSecondDeck buttons which were created to be displayed visibly on the tableComponent.

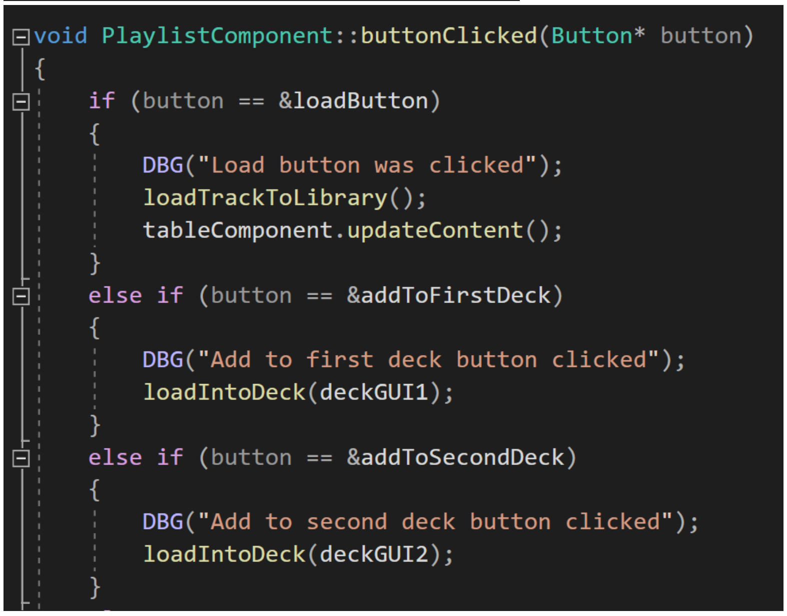


```
void PlaylistComponent::resized()
{
    //set bounds of for child components that your component contains

    tableComponent.setBounds(0, 0, getWidth(), getHeight());
    loadButton.setBounds(0, getWidth()/4, getWidth()/4, getHeight()/4);
    searchTracks.setBounds(getWidth() / 4, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToFirstDeck.setBounds(getWidth() / 2, getWidth() / 4, getWidth() / 4, getHeight() / 4);
    addToSecondDeck.setBounds(getWidth() / 4*3, getWidth() / 4, getWidth() / 4, getHeight() / 4);
}
```

In the resized() function, setBounds is used to create a size and assign a location for the addToFirstDeck and addToSecondDeck buttons in the tableComponent.

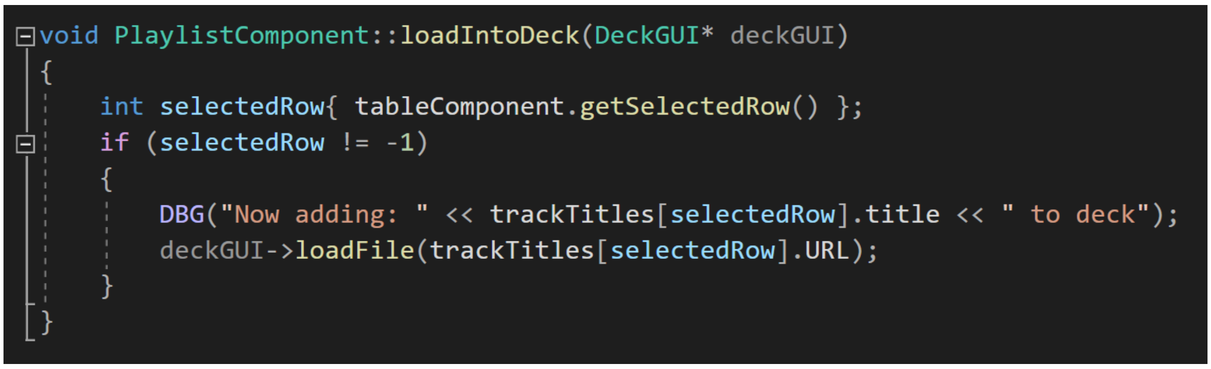


```
//create button listeners
loadButton.addListener(this);
addToFirstDeck.addListener(this);
addToSecondDeck.addListener(this);
```

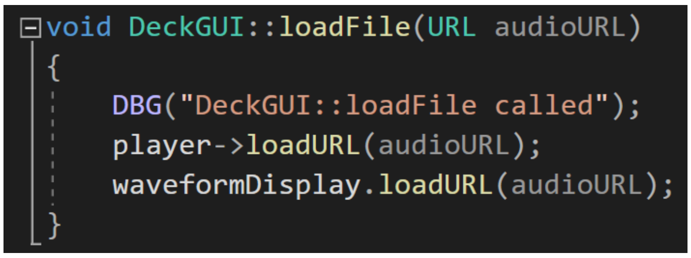


```
void PlaylistComponent::buttonClicked(Button* button)
{
    if (button == &loadButton)
    {
        DBG("Load button was clicked");
        loadTrackToLibrary();
        tableComponent.updateContent();
    }
    else if (button == &addToFirstDeck)
    {
        DBG("Add to first deck button clicked");
        loadIntoDeck(deckGUI1);
    }
    else if (button == &addToSecondDeck)
    {
        DBG("Add to second deck button clicked");
        loadIntoDeck(deckGUI2);
    }
```

The code above adds a button listener to the addToFirstDeck and addToSecondDeck buttons such that its ties the event of clicking on the button to the loadIntoDeck() function, which will be explained later. The deckGUI component will then be updated as new tracks are loaded from the tableComponent and into it.



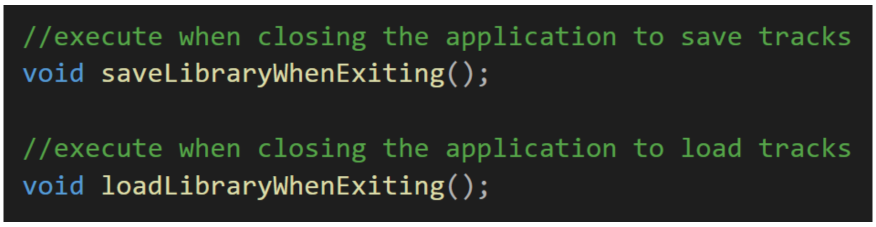
```
void PlaylistComponent::loadIntoDeck(DeckGUI* deckGUI)
{
    int selectedRow{ tableComponent.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Now adding: " << trackTitles[selectedRow].title << " to deck");
        deckGUI->loadFile(trackTitles[selectedRow].URL);
    }
}
```



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

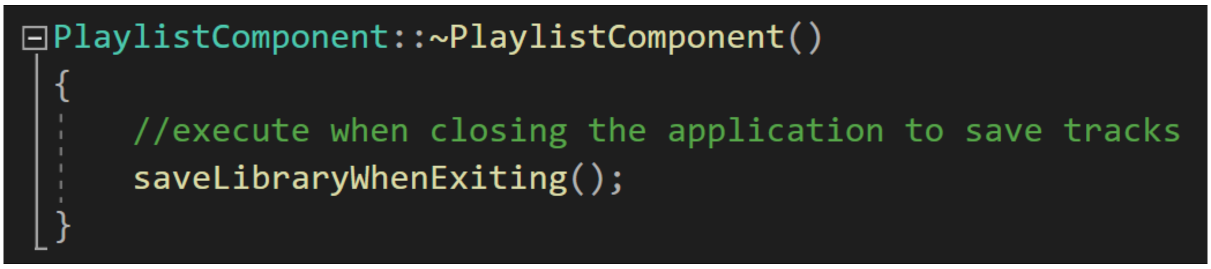
In the loadIntoDeck() function, when the user selects a row in the tableComponent, the getSelectRow() function passes it into a new integer variable called selected row. The URL of the corresponding track from the selectedRow will pass through the loadFile function, which can be found in the DeckGUI class. The loadFile() function loads the URL into the player to create the waveform display which will then be loaded and displayed in the DeckGUI component. Hence, the files can be loaded into the deck.

R3E: The music library persists so that it is restored when the user exits then restarts the application



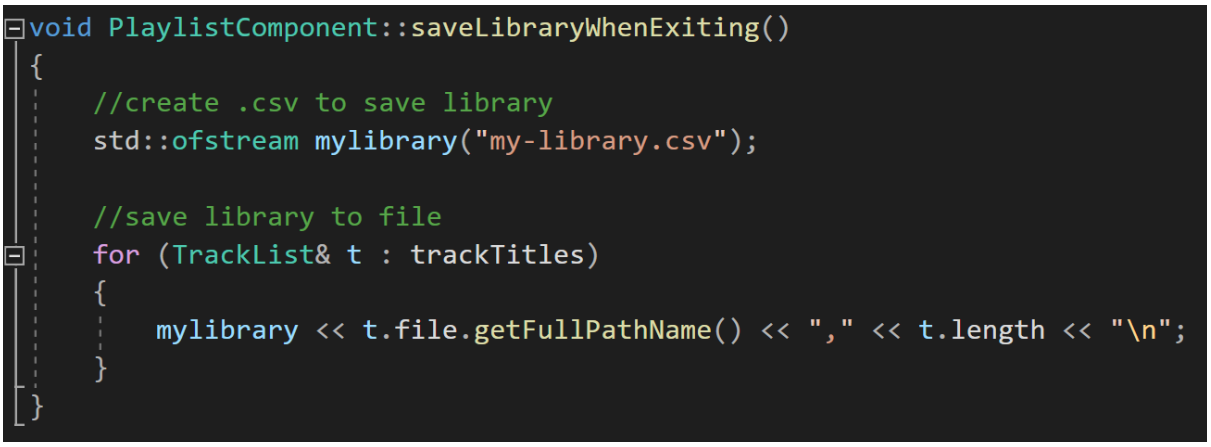
```
//execute when closing the application to save tracks
void saveLibraryWhenExiting();

//execute when closing the application to load tracks
void loadLibraryWhenExiting();
```

The functions saveLibraryWhenExiting() and loadLibraryWhenExiting() are implemented in the header file of the PlaylistComponent class so that the tableComponent can be stored and loaded when the application is closed and reopened.



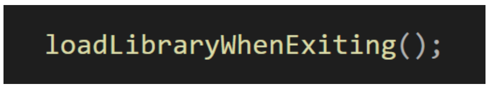
```
PlaylistComponent::~PlaylistComponent()
{
    //execute when closing the application to save tracks
    saveLibraryWhenExiting();
}
```

saveLibraryWhenExiting() function is called in the deconstructor of the PlaylistComponent class, which will execute when the application is closed, to save music library data.



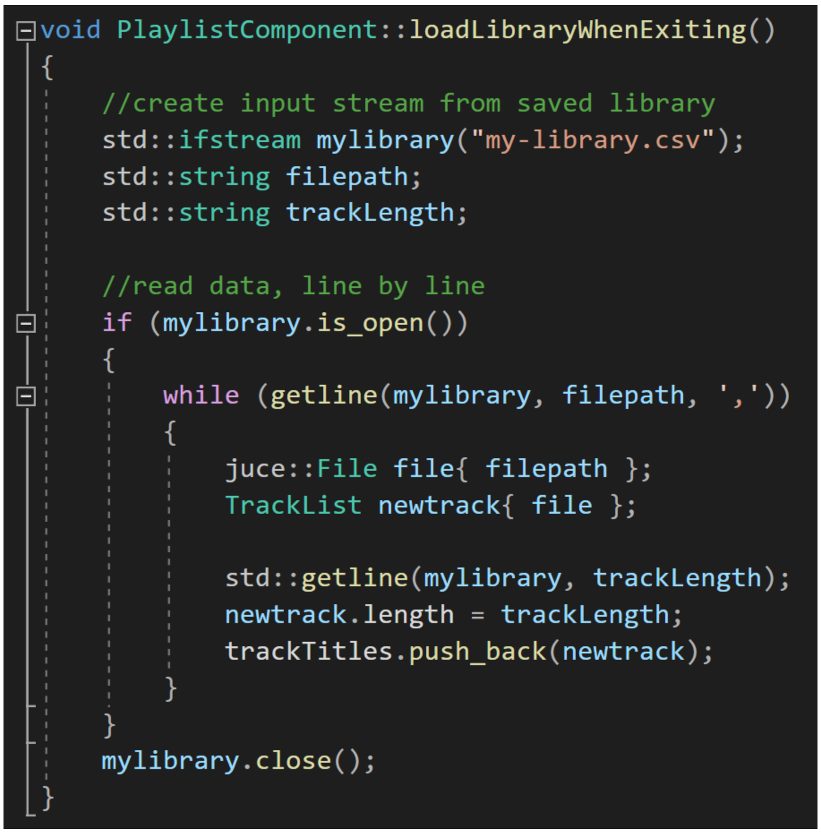
```
void PlaylistComponent::saveLibraryWhenExiting()
{
    //create .csv to save library
    std::ofstream mylibrary("my-library.csv");

    //save library to file
    for (TrackList& t : trackTitles)
    {
        mylibrary << t.file.getFullPathName() << "," << t.length << "\n";
    }
}
```

In the saveLibraryWhenExiting() function, the class ofstream is used to create a .csv file called “my-library.csv”, which is in output file stream. For each track object in the tableComponent, the .csv file saves the path and track length by using functions getFullPathName() and length(). Hence, the library is saved whenever the application is closed.



```
loadLibraryWhenExiting();
```

The loadLibraryWhenExiting() function is called in the constructor of the PlaylistComponent class such that it loads the library data when the application is opened.



```
void PlaylistComponent::loadLibraryWhenExiting()
{
    //create input stream from saved library
    std::ifstream mylibrary("my-library.csv");
    std::string filepath;
    std::string trackLength;

    //read data, line by line
    if (mylibrary.is_open())
    {
        while (getline(mylibrary, filepath, ','))
        {
            juce::File file{ filepath };
            TrackList newtrack{ file };

            std::getline(mylibrary, trackLength);
            newtrack.length = trackLength;
            trackTitles.push_back(newtrack);
        }
    }
    mylibrary.close();
}
```

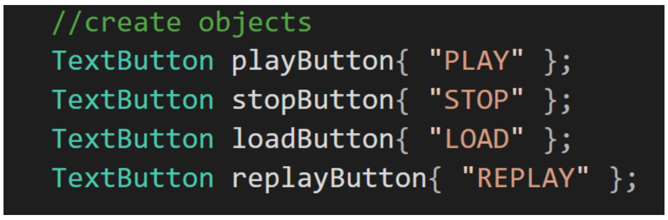
In the loadLibraryWhenExiting() function, the .csv file called “my-library.csv” is being opened as an input file stream to create filepath and trackLength string variables. If the library is opened, data from the .csv file is being read line by line as getline() is used to extract characters until a comma is reached. A new file is created to store the filepath. A TrackList object called newtrack will the store that filepath. The getline() function then uses the variable trackLength to extract the duration of each track in the .csv file, and store it as the length in each newtrack object. The object newtrack is then pushed backed and loaded into trackTitles before the .csv file closes its data.

R4: Implementation of a complete custom GUI

R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls

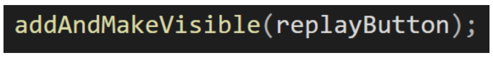
For this requirement, I have chosen to add two controls to the DeckGUI, which are the replay and delete button. They function by replaying the tracks in the deck component and by deleting tracks in the playlist component respectively.

REPLAY CONTROLS



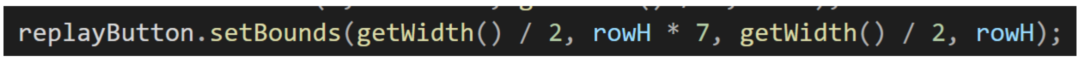
```
    //create objects
    TextButton playButton{ "PLAY" };
    TextButton stopButton{ "STOP" };
    TextButton loadButton{ "LOAD" };
    TextButton replayButton{ "REPLAY" };
```

The figure above shows how text button objects are created in the header file which in turn creates the replay button.



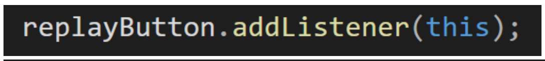
```
addAndMakeVisible(replayButton);
```

The following code above which can be found in the cpp file allows the replay button created to be displayed visibly on the audio deck.

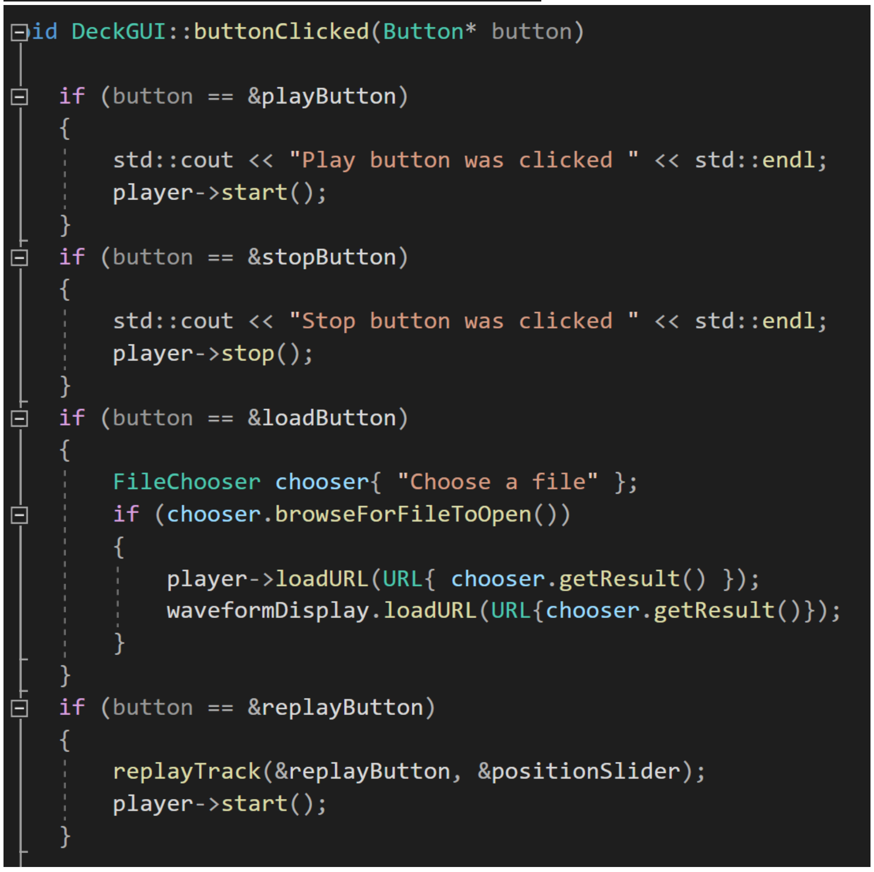


```
replayButton.setBounds(getWidth() / 2, rowH * 7, getWidth() / 2, rowH);
```

In the resized() function, setBounds is used to create a size and assign a location for the replay button on the audio player deck.

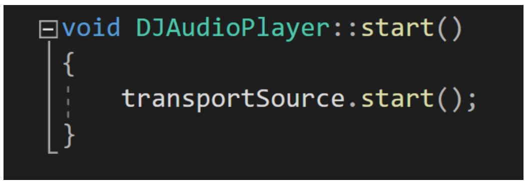


```
replayButton.addListener(this);
```



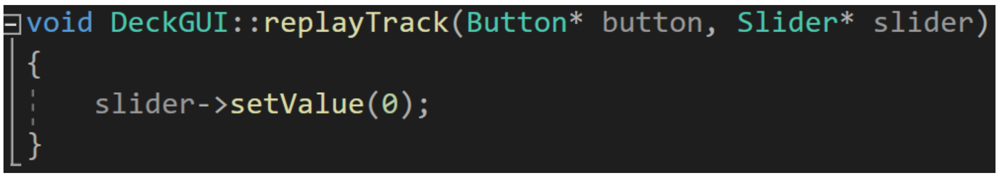
```
id DeckGUI::buttonClicked(Button* button)

    if (button == &playButton)
    {
        std::cout << "Play button was clicked " << std::endl;
        player->start();
    }
    if (button == &stopButton)
    {
        std::cout << "Stop button was clicked " << std::endl;
        player->stop();
    }
    if (button == &loadButton)
    {
        FileChooser chooser{ "Choose a file" };
        if (chooser.browseForFileToOpen())
        {
            player->loadURL(URL{ chooser.getResult() });
            waveformDisplay.loadURL(URL{chooser.getResult()});
        }
    }
    if (button == &replayButton)
    {
        replayTrack(&replayButton, &positionSlider);
        player->start();
    }
```

The code above adds a button listener to the replayButton such that its ties the event of clicking on the button to the replayTrack() function, which will be elaborated further later, taking in the positionSlider as its paramenters. Also, the start() function in the DJAudioPlayer class is called and loaded into the player, allowing the deckGUI to restart the current track.



```
void DJAudioPlayer::start()
{
    transportSource.start();
}
```

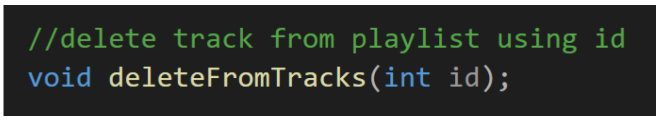
In the DJAudioPlayer class, the start() function as shown above works by allowing the object transportSource, which is the audio file, to start playing.



```
void DeckGUI::replayTrack(Button* button, Slider* slider)
{
    slider->setValue(0);
}
```

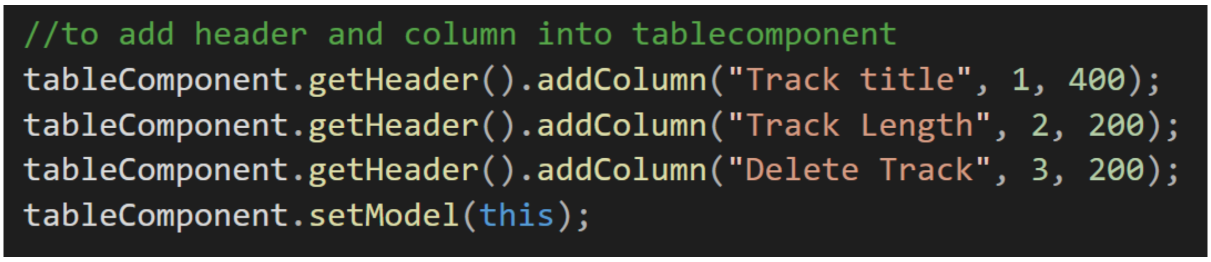
In the DeckGUI class, the replayTrack() function works, by setting the slider value back to 0, effectively restarting the track. Thus, the DeckGUI has a new control added, which is to allow tracks to be replayed.

DELETE CONTROLS



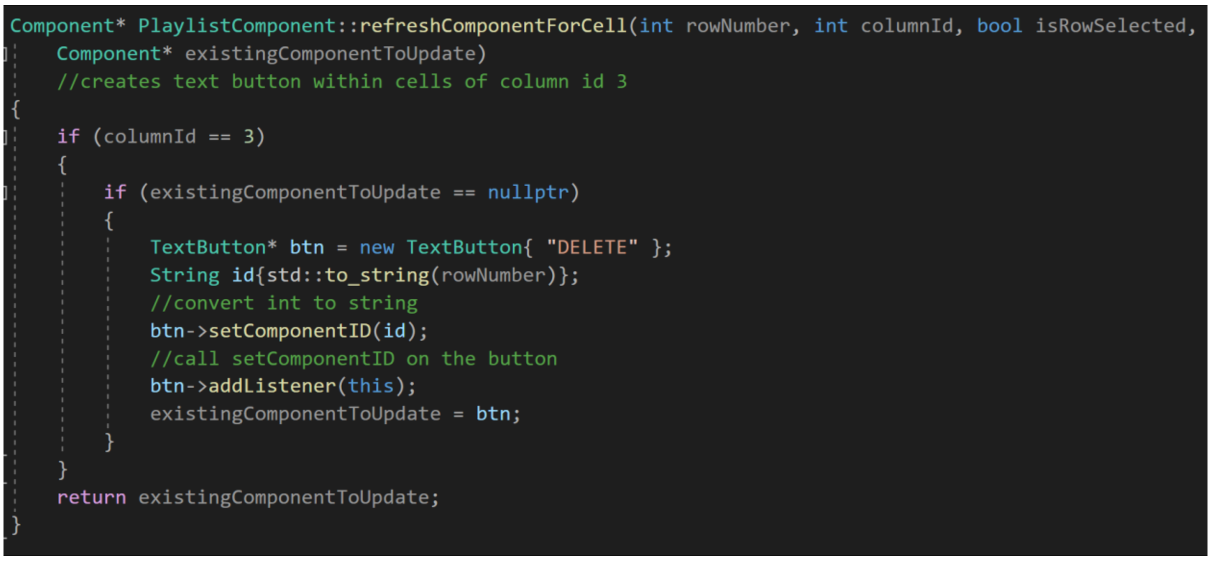
```
//delete track from playlist using id
void deleteFromTracks(int id);
```

The function deleteFromTrack() is implemented in the header file of the PlaylistComponent class so that individual tracks in the tableComponent can be deleted away.



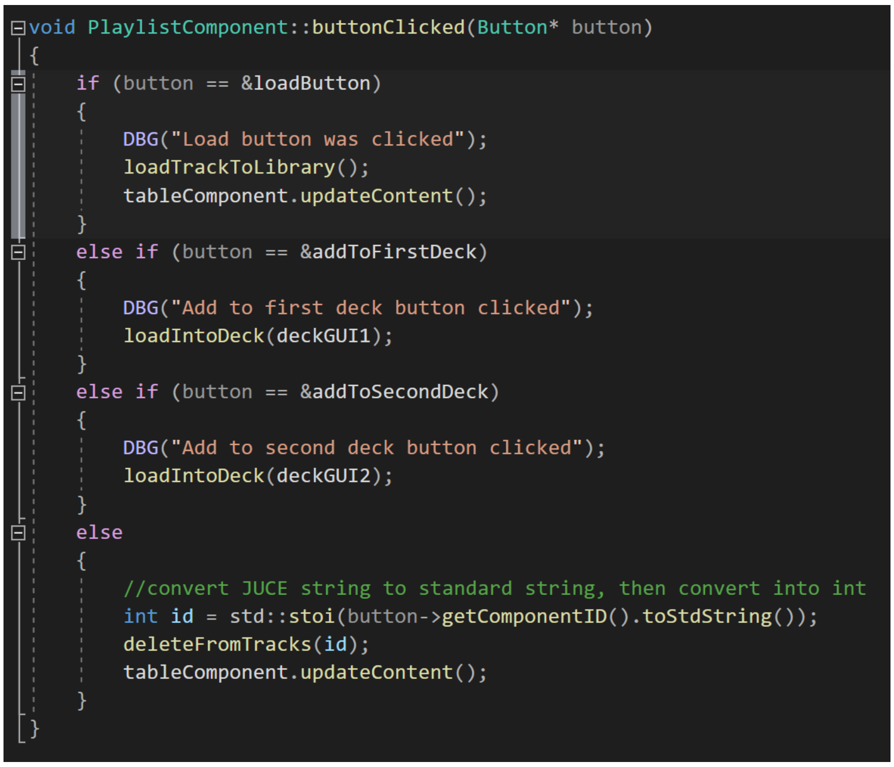
```
//to add header and column into tablecomponent
tableComponent.getHeader().addColumn("Track title", 1, 400);
tableComponent.getHeader().addColumn("Track Length", 2, 200);
tableComponent.getHeader().addColumn("Delete Track", 3, 200);
tableComponent.setModel(this);
```

In the PlaylistComponent class, the code above is used to add the header and column into the tableComponent. Hence, in the third column, the string "Delete Track" is passed into the cell, the columnId is set to 3, and the length of the delete button is 200px.

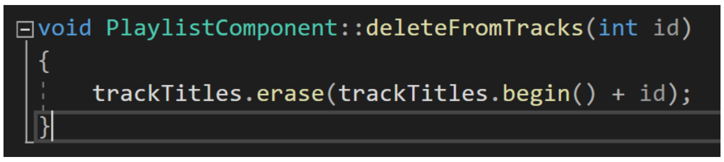


```
Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int columnId, bool isRowSelected,
    Component* existingComponentToUpdate)
    //creates text button within cells of column id 3
{
    if (columnId == 3)
    {
        if (existingComponentToUpdate == nullptr)
        {
            TextButton* btn = new TextButton{ "DELETE" };
            String id{std::to_string(rowNumber)};
            //convert int to string
            btn->setComponentID(id);
            //call setComponentID on the button
            btn->addListener(this);
            existingComponentToUpdate = btn;
        }
    }
    return existingComponentToUpdate;
}
```

In the refreshComponentForCell() function in the PlaylistComponent, where the columnId equals to 3, a TextButton object named btn will be created. A string is assigned to the button such that it is labelled as a "DELETE" button. rowNumber is passed into the String id, which is initially converted from an integer to a string. Additionally, setComponentID and the addListener button is added to the object btn. The function then returns the value of the btn.



```cpp
void PlaylistComponent::buttonClicked(Button* button)
{
    if (button == &loadButton)
    {
        DBG("Load button was clicked");
        loadTrackToLibrary();
        tableComponent.updateContent();
    }
    else if (button == &addToFirstDeck)
    {
        DBG("Add to first deck button clicked");
        loadIntoDeck(deckGUI1);
    }
    else if (button == &addToSecondDeck)
    {
        DBG("Add to second deck button clicked");
        loadIntoDeck(deckGUI2);
    }
    else
    {
        //convert JUCE string to standard string, then convert into int
        int id = std::stoi(button->getComponentID().toStdString());
        deleteFromTracks(id);
        tableComponent.updateContent();
    }
}
```



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

The buttonClicked() function is required to link the event of clicking the button to deleting the corresponding tracks from tableComponent. In the function, the integer variable id is initialised by converting the button id from juce string to standard string, before converting to an integer and storing the value. The function deleteFromTracks() is then called. It uses erase() to delete tracks in the inherited id + 1 position. Going back to buttonClicked(), the tableComponent is updated using the updateContent() function after the track object is removed. Hence, functional delete buttons are implemented into the application.

R4B: GUI layout includes the custom Component from R2

The customization that I have chosen to implement for the deck control component is to change all three sliders for volume, speed, and position to rotary sliders, as well as attaching labels to the respective rotary sliders. This has already been covered in the sub-requirement for R2A, which requested for custom graphics to be implemented in the paint function.

```
volumeSlider.setSliderStyle(Slider::SliderStyle::Rotary);
speedSlider.setSliderStyle(Slider::SliderStyle::Rotary);
positionSlider.setSliderStyle(Slider::SliderStyle::Rotary);
```

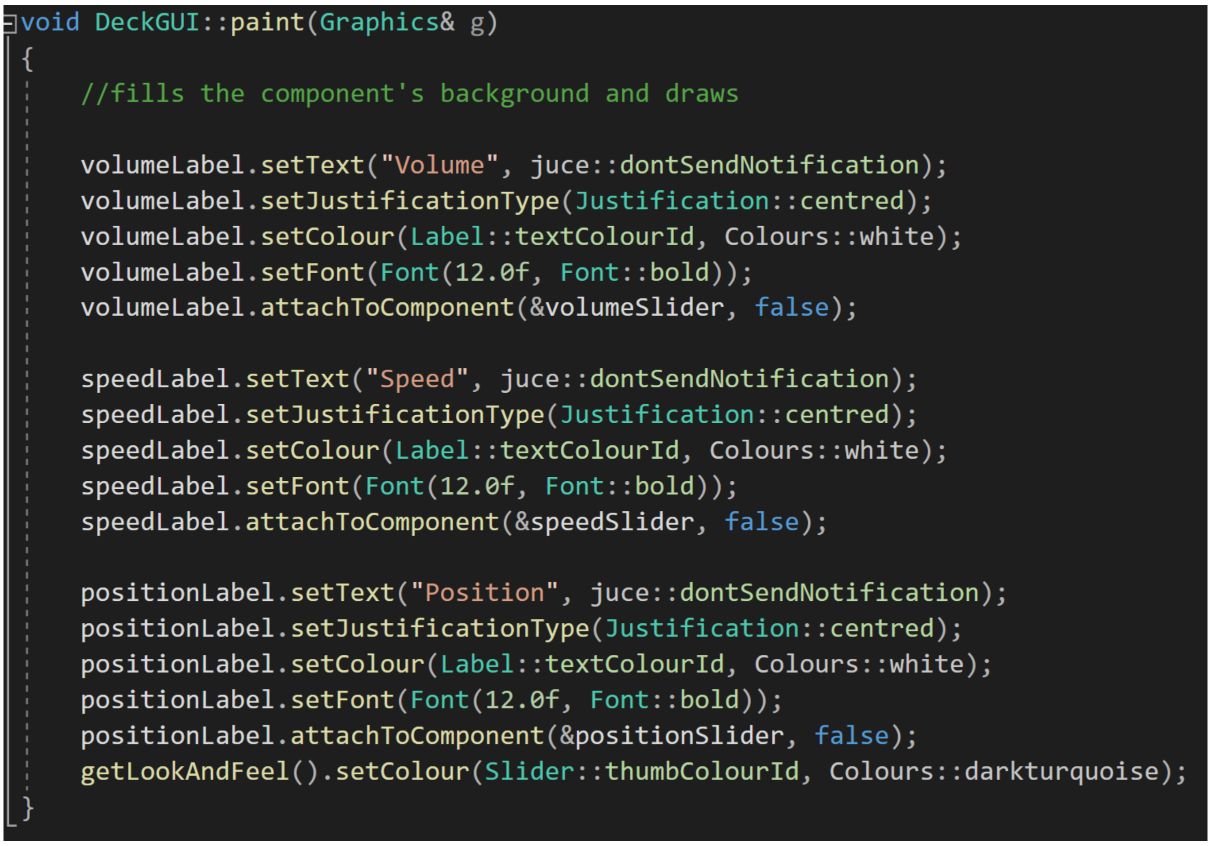
In the DeckGUI cpp file, the function setSliderStyle() is used to set the slider style of all three volume, speed and position slider objects to rotary sliders by inheriting the SliderStyle from the slider class. Hence, the volume, speed and position sliders are of rotary type.

```
Label volumeLabel;
Label speedLabel;
Label positionLabel;
```

In the DeckGUI header file, the objects volumeLabel, speedLabel and positionLabel are initialised. They are inherited from the juce library using the label class.

```
addAndMakeVisible(volumeLabel);
addAndMakeVisible(speedLabel);
addAndMakeVisible(positionLabel);
```

In the DeckGUI cpp file, the addAndMakeVisible() function allows all three label objects, volumeLabel, speedLabel and positionLabel to be displayed visibly on the audio deck.



```
void DeckGUI::paint(Graphics& g)
{
    //fills the component's background and draws

    volumeLabel.setText("Volume", juce::dontSendNotification);
    volumeLabel.setJustificationType(Justification::centred);
    volumeLabel.setColour(Label::textColourId, Colours::white);
    volumeLabel.setFont(Font(12.0f, Font::bold));
    volumeLabel.attachToComponent(&volumeSlider, false);

    speedLabel.setText("Speed", juce::dontSendNotification);
    speedLabel.setJustificationType(Justification::centred);
    speedLabel.setColour(Label::textColourId, Colours::white);
    speedLabel.setFont(Font(12.0f, Font::bold));
    speedLabel.attachToComponent(&speedSlider, false);

    positionLabel.setText("Position", juce::dontSendNotification);
    positionLabel.setJustificationType(Justification::centred);
    positionLabel.setColour(Label::textColourId, Colours::white);
    positionLabel.setFont(Font(12.0f, Font::bold));
    positionLabel.attachToComponent(&positionSlider, false);
    getLookAndFeel().setColour(Slider::thumbColourId, Colours::darkturquoise);
}
```

The above snippet of code is taken from the paint() function in the DeckGUI class. The paint() function is used to render our component on the application interface. For each label, I used the function setText() to take in a string parameter which will be displayed on the application as the name of each slider. They are named "Volume", "Speed", and "Position" for their respective slider with the same namesake. setText() also takes in another parameter which is set to not push any notification to users. setJustificationType() function is used to set the alignment of each label such that it is in the center of each slider. setColour() function takes in a textColourId integer and also paints all three labels white using the juce class colours. The setFont() function takes in font size and font type as parameter. The label sliders are set to have a font size of 12.0f, and are emboldened. Lastly, each slider object uses the attachToComponent() function to attach to their respective sliders, and takes in a false Boolean parameter to represent that they are attached to the top of their respective slider, position-wise.

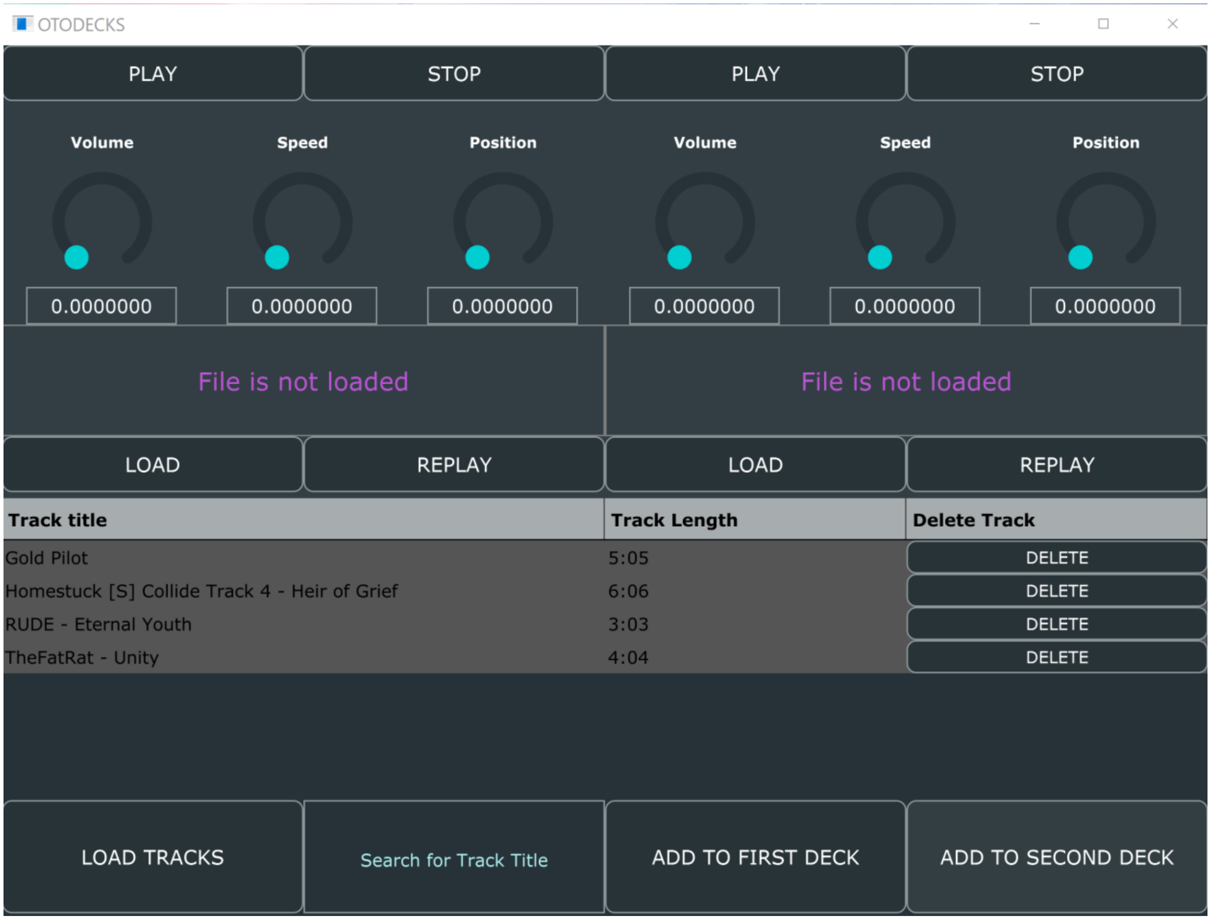
In the last line of code, the setColour() function is called such that the thumb color of all rotary sliders is set to dark turquoise.

R4C: GUI layout includes the music library component from R3

The music library component, named tableComponent, is created due to the codes from both the DeckGUI class and the PlaylistComponent class. Each functionality that are added are positioned such that they have a specific GUI layout, and the explanation and positioning for each of these functions or components can be found in the explanation for R3 and R4.

To recap, the GUI layout for the music library includes a load button, search functionality, add to first and second deck buttons, as well as delete buttons. These were created, customized, and explained previously in R3A, R3C, R3D, and R4A respectively.

Additionally, the playlist is created such that it shows the track title and track length as well. This was also previously explained in R3B.



Track title	Track Length	Delete Track
Gold Pilot	5:05	DELETE
Homestuck [S] Collide Track 4 - Heir of Grief	6:06	DELETE
RUDE - Eternal Youth	3:03	DELETE
TheFatRat - Unity	4:04	DELETE