Coding workshop: Refactoring the audio playback code into a class

Introduction

In this worksheet, we will refactor the audio player functionality into a new class called DJAudioPlayer. Part of this process involves inheriting from the AudioSource class, which is an abstract class that demands we implement certain pure virtual functions. Once the audio playback code is in its own class, we can create multiple instances of that class and have more than one player.

Refactoring the audio playing code

Let's begin by thinking about what exactly we'd like to see of DJAudioPlayer from the outside. Since we are the programmer, we get to choose what we want our audio player to do. For this worksheet, I just want the following features:

- loadURL
- play
- stop
- setPosition
- setGain

To add a new class to our project, we need to use Projucer.

Adding the new class to the project with Projucer:

In Projucer, with your project loaded, click on the view menu -> show file explorer panel option. This should show the list of files in your project, current consisting of:

Right-click where it says source on the left panel and select 'add new CPP and header file'. Call the new class DJAudioPlayer. This will create new files and add them to the project. Now save the project (*very important step!*) from Projucer, and open it up in your IDE.

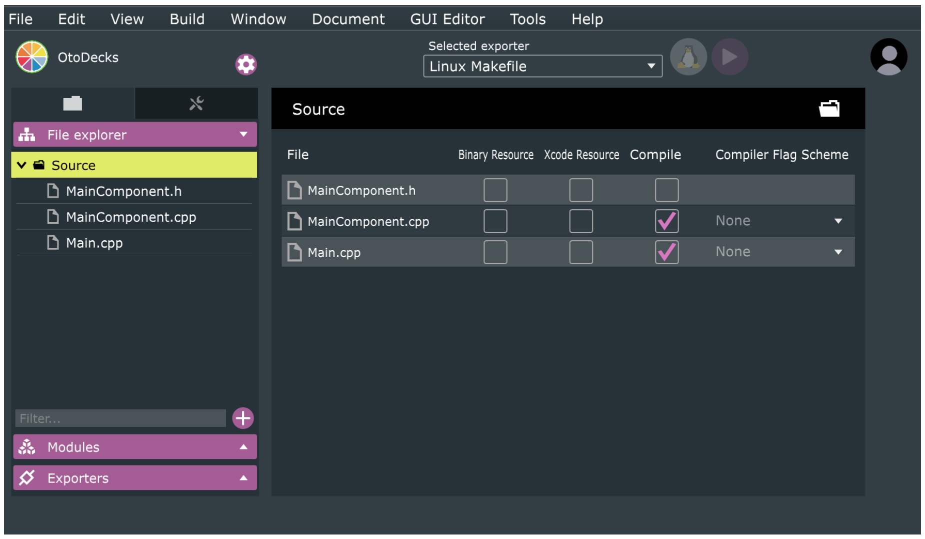
The DJAudioPlayer public interface

Now we are ready to define the public interface for the DJAudioPlayer class. This is what other objects can see of DJAudioPlayer.

In the DJAudioPlayer header file, add this code:

```
#pragma once

// assumes your JuceHeader file is here
// check your MainComponent.h to see
// how the include is configured on your
```



Selected exporter: Linux Makefile

Source

File	Binary Resource	Xcode Resource	Compile	Compiler Flag Scheme
MainComponent.h	☐	☐	☐	
MainComponent.cpp	☐	☐	✓	None
Main.cpp	☐	☐	✓	None

Figure 1: Projucer file explorer

```
// system
#include "../JuceLibraryCode/JuceHeader.h"

class DJAudioPlayer
{
    public:
        DJAudioPlayer();
        ~DJAudioPlayer();
        void loadURL(URL file);
        void play();
        void stop();
        void setPosition(double posInSecs);
        void setGain(double gain);
};
```

Comment the code

Now you need to write a sensible comment above each function in the public interface explaining what it does. Here is an example:

```
/** start playing the file */
void play();
```

Go ahead and add comments to the other function members of the DJAudioPlayer class.

Write out a framework in the CPP file

Now let's put in a compilable framework for the class in the DJAudioPlayer.cpp file.

```cpp
#include "DJAudioPlayer.h"

DJAudioPlayer::DJAudioPlayer()
{

}

DJAudioPlayer::~DJAudioPlayer()
{

}

void DJAudioPlayer::loadURL(URL audioURL)
{

}

void DJAudioPlayer::play()
{

}

void DJAudioPlayer::stop()
{

}

void DJAudioPlayer::setPosition(double posInSecs)
{

}

void DJAudioPlayer::setGain(double gain)
{

}
```

Add a DJAudioPlayer to MainComponent to trigger linking

Now we have a compilable DJAudioPlayer class, let's add a DJAudioPlayer object as a data member to the MainComponent class. Go into MainComponnent.h and include the DJAudioPlayer header:

```
#include "DJAudioPlayer.h"
```

Then add this line to the private section of the MainComponent class:

```
DJAudioPlayer player1;
```

Compile the project and run it to confirm things are working ok.

Implement the loadURL function

The first step is implementing the loadURL function that allows the DJAudioPlayer to be told to load a file. We'll need some JUCE audio classes to do that. Add this to your DJAudioPlayer header file, inside the private section of the class:

```
private:
    AudioFormatManager formatManager;
    std::unique_ptr<AudioFormatReaderSource> readerSource;
    AudioTransportSource transportSource;
```

Now go into the DJAudioPlayer cpp file and add this to the constructor:

```
formatManager.registerBasicFormats();
```

That makes sure the audio system knows about a set of basic audio formats such as WAV and MP3.

Now in the loadURL function in DJAudioPlayer.cpp:

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

Note this is very similar to the code we previously had in the MainComponent constructor, but it uses an URL sent in as a parameter. You can go ahead and comment out MainCompoment's loadURL function's contents now. We will not be using it any more!

Test loadURL by hooking it into the load button

Now we should check the loadURL function works. We are going to hook it into the load button on the main app. Change the event listener code in MainComponent::buttonClicked to call load on the player.

```
if (button == &loadButton )
{
  FileChooser chooser ("Select a Wave file to   play...");
  if (chooser.browseForFileToOpen())
  {
      URL audioURL = URL{chooser.getResult()};
      player1.loadURL(audioURL);
  }
}
```

Now run it and see if the flow goes through to the loadURL function on DJAudioPlayer as you'd expect. Add a cout print to loadURL to check the message gets through.

Implement play

Now let's hook up the play button event listener to call play on the DJAudioPlayer. In MainComponent::buttonClicked, call play on the player object:

```
if (button == &playButton )
{
  playing = true;
  player1.play();
}
```

Then into DJAudioPlayer::play, call start on the transportSource:

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

Inherit from AudioSource and implement pure virtual functions

Now for the clever bit - we are going to change our DJAudioPlayer class into a JUCE AudioSource so we can easily ask it for audio from the MainComponent app. Otherwise, how will we get audio out of the transportSource and into the audio buffer in the MainComponent?

Set up the inheritance relationship

Go into DJAudioPlayer.h and add an inheritance relationship to the class:

```
class DJAudioPlayer : public AudioSource {
```

Now we are inheriting from AudioSource, we must provide an implementation for certain functions from the AudioSource class as they are pure virtual in AudioSource. Check out the signatures from the AudioSource class:

```
virtual void prepareToPlay (int samplesPerBlockExpected, double sampleRate)=0

virtual void releaseResources ()=0

virtual void getNextAudioBlock (const AudioSourceChannelInfo &bufferToFill)=0
```

Note that these functions are declared as virtual, and they have =0 assigned for their implementation. The =0 makes them pure virtual, so we have to implement them if we want to create a 'concrete' class. The textbook covers pure virtual functions in Chapter 14, p561.

Implement the pure virtual functions

Now we have to implement the audio source interface in our class. Put this into the public section of DJAudioPlayer.h:

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

Then in the DJAudioPlayer.cpp, put the framework in:

```
void DJAudioPlayer::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
}
void DJAudioPlayer::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
}
void DJAudioPlayer::releaseResources()
{
}
```

Do a quick compile to check things are working ok.

Write working audio generation code

Next, we need to fill out those audio functions so they pull audio from the transportSource variable, which should be ready to play the audio file after loadURL.

Update the audio functions in the DJAudioPlayer.cpp file:

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

void DJAudioPlayer::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    if (readerSource.get() == nullptr)
    {
        bufferToFill.clearActiveBufferRegion();
        return;
    }
    transportSource.getNextAudioBlock (bufferToFill);
}

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

Some notes:

- getNextAudioBlock exactly does what it did when it was in the MainComponent class. In essence, it checks if the readerSource is ready (remember the reader sits underneath the transportSource and does the low-level file reading).
- if the readerSource is ready, we just call getNextAudioBlock on the transportSource and pass it the buffer.

Connect through the the MainComponent getNextAudioBlock function

MainComponent is currently in charge of generating the audio output, so we need it to call through to the DJAudioPlayer to get the audio. In MainComponent.cpp:

```
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    player1.getNextAudioBlock(bufferToFill);
}
```

We are just passing the bufferToFill through to our DJAudioPlayer class and assuming it knows what to do. We might call this a delegation relationship - the MainComponent class delegates responsibility for audio generation to the DJAudioPlayer class.

Compile and test!

Connecting the other audio functions

We need to connect releaseResources and prepareToPlay through from MainComponent to DJAudioPlayer as well. In MainComponent.cpp:

```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
}
```

```
void MainComponent::releaseResources()
{
    player1.releaseResources();
}
```

That's it for the basic audio implementation - we should be able to load a file and play it, all using the DJAudioPlayer class.

Implement stop

Now let's complete the playback controls. Back to the event listener code in MainComponent.cpp:

```
if (button == &stopButton )
{
    player1.stop();
}
```

Then into DJAudioPlayer.cpp:

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

Compile and test that you can now start and stop the audio using the buttons.

Implement setPosition and setGain

Now we can implement setPosition and setGain. In DJAudioPlayer.cpp:

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

void DJAudioPlayer::setGain(double gain)
{
  transportSource.setGain(gain);
}
```

Since setPosition takes and argument in seconds, we can make the code safer by checking the incoming value:

```
void DJAudioPlayer::setPosition(double posInSecs)
{
  if (posInSecs < 0 || posInSecs > transportSource.getLengthInSeconds()){
    std::cout << "DJAudioPlayer::setPosition: warning set position " << posInSecs \
    << " greater than length " << transportSource.getLengthInSeconds() << std::endl;
    return;
  }
  transportSource.setPosition(posInSecs);
}
```

There is an interesting question here - where is the best place to report errors? Let's say we implemented a GUI text entry box that allows the user to set the position. If they enter an invalid position, we should tell them via the GUI. Therefore, we would need to check the position in the GUI layer, not at the low-level DJAudioPlayer. The warning printed out above is really for developers, not end-users. How would you handle this?

An alternative approach for setPosition

You might want to present a more straightforward interface for position control from your DJAudioPlayer class. At the moment, the setPosition function requires that we know how long the audio file is in seconds from the calling class (MainComponent) but how does MainComponent know how long the audio file is, given that DJAudioPlayer is now managing all the audio work?

It would be easier if we could send in a value between 0 and 1, then have the DJAudioPlayer convert that into seconds.

Add this to DJAudioPlayer.h, in the public section of the class definition:

```
void setPositionRelative(double pos);
```

Then the implementation:

```
void DJAudioPlayer::setPositionRelative(double pos)
{
  double posInSecs = pos * transportSource.getLengthInSeconds();
  setPosition(posInSecs);
}
```

Notes:

- We assume they sent in a value between 0 and 1
- We scale it into the range 0-length in seconds
- We do not interact with the transportSource directly, we pass that off to our original setPosition function, which then does the checking on the range of the parameter.

Clean up

We can now go ahead and remove some of the un-needed audio playback code from MainComponent.h. Remove these fields from the private section in MainComponent.h:

```
AudioFormatManager formatManager;
std::unique_ptr<AudioFormatReaderSource> readerSource;
AudioTransportSource transportSource;
```

Then check through MainComponent.cpp, removing any reference to these fields. You could just try and compile after removing the fields from the header and then fix the errors, but it might be easier to search through the code for those variable names. Places to look:

- The constructor does not need to do any audio work now
- You can clear out any commented out code in getNextAudioBlock - it can just be this:

  void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill) { player1.getNextAudioBlock(bufferToFill); }

Complete the implementation yourself:

To complete the implementation, go ahead and add a slider to control the position.

Experiment - can you add another DJAudioPlayer data member to your MainComponent and control the two players with a set of buttons?

Conclusion

In this worksheet we have refactored the audio file code into a new class. This class abstracts away the complexity of playing audio files, and makes our MainComponent class much neater.

The complete DJAudioPlayer header file

Here is the complete DJAudioPlayer.h file, noting that there might be small variations from your one.

```
#pragma once
// assumes your JuceHeader file is here
// check your MainComponent.h to see
// how the include is configured on your
// system
#include "../JuceLibraryCode/JuceHeader.h"

class DJAudioPlayer : public AudioSource {
public:

    DJAudioPlayer();
    ~DJAudioPlayer();

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

    void loadURL(URL audioURL);
    void setGain(double gain);
    void setSpeed(double ratio);
    void setPosition(double posInSecs);
    void setPositionRelative(double pos);


    void play();
    void stop();

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

};
```

The complete DJAudioPlayer CPP file

Here is the complete DJAudioPlayer.cpp file, noting that there might be small variations from your one.

```
#include "DJAudioPlayer.h"

DJAudioPlayer::DJAudioPlayer()
{
    formatManager.registerBasicFormats();
}
DJAudioPlayer::~DJAudioPlayer()
{

}

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();
}

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

}
void DJAudioPlayer::setSpeed(double ratio)
{
if (ratio < 0 || ratio > 100.0)
    {
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::enc
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
void DJAudioPlayer::setPosition(double posInSecs)
{
    transportSource.setPosition(posInSecs);
}

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


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