AUDIO C++

Level	Intermediate
Source	
Status	Ongoing
Related to Coding Projects (Files)	
Tags	

JUCE FRAMEWORK:

- EQ PROYECT
  - SETUP NOTES
    - PluginProcessor.h

      In a JUCE plugin, there are two functions that are the most important:

```
// PREPARE TO PLAY FUNCTION:
   void prepareToPlay (double sampleRate, int samplesPerBlock) override;
// Gets called by the host when it's about to start playback


// PROCESS BLOCK FUNCTION:
   void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;
// When play button is activated in transport control.
// The host starts sending buffers at a regular rate into the plugin.
// Its the plugin's job to give back any finished audio that is done processing.
/// DO NOT INTERRUPT THE PROCESSBLOCK OR ANYTHING THAT GOES INSIDE IT \
```

      IMPORTANT: FIGURE OUT A WAY TO MAKE THE CODE RUN IN A CERTAIN TIMESPAN.

    - PluginProcessor.cpp

```
//PREPARE TO PLAY
void SimpleEQAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
}
//PROCESSBLOCK
void SimpleEQAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    // In case we have more outputs than inputs, this code clears any output
```

```
    // channels that didn't contain input data, (because these aren't
    // guaranteed to be empty - they may contain garbage).
    // This is here to avoid people getting screaming feedback
    // when they first compile a plugin, but obviously you don't need to keep
    // this code if your algorithm always overwrites all the output channels.
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // This is the place where you'd normally do the guts of your plugin's
    // audio processing...
    // Make sure to reset the state if your inner loop is processing
    // the samples and the outer loop is handling the channels.
    // Alternatively, you can process the samples with the channels
    // interleaved by keeping the same state.
    for (int channel = 0; channel < totalNumInputChannels; ++channel)
    {
        auto* channelData = buffer.getWritePointer (channel);

        // ..do something to the data...
    }
}
```

▼ CREATING AUDIO PARAMETERS

▼ About & Parameter Creation

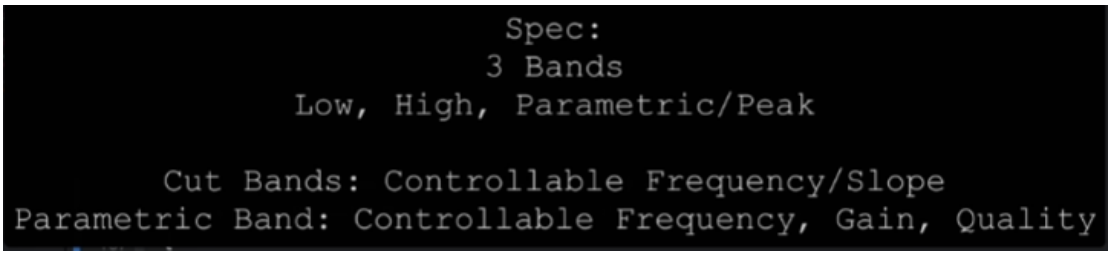
Audio plugins depend on parameters to control the various parts of the DSP. JUCE *uses an object called the* `AudioProcessorValueTreeState` *to coordinate syncing these parameters with the knobs on the GUI and the variables on the processor.*

```
// In Pluginprocessor.h > declare apvts

static juce::AudioProcessorValueTreeState::ParameterLayout
  createParameterLayout();
juce::AudioProcessorValueTreeState apvts /*name*/
{
  *this, nullptr, "Parameters", createParameterLayout()
};
*this: //processor to connect to
nullptr: //no undo manager
"Parameters" // const identifier

// APVTS EXPECTS TO PROVIDE A LIST OF ALL PARAMETERS WHEN
// ITS CREATED >> APVTS PARAMETERLAYOUT
```

- Declaring create apvts parameter layout in PluginProcessor.cpp:



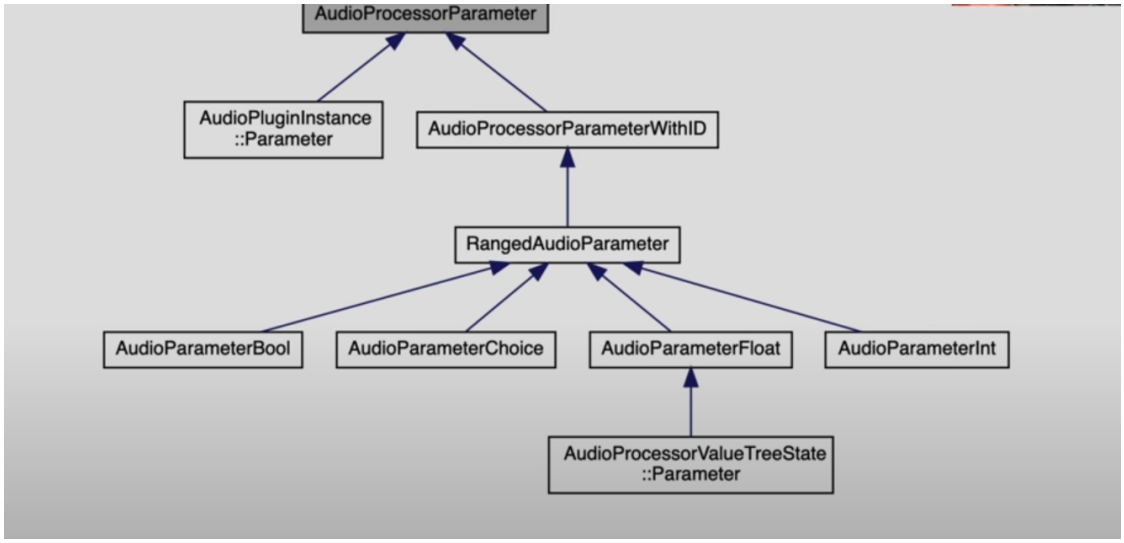

```
// Keeping DSP and GUI simple:
juce::AudioProcessorValueTreeState::ParameterLayout
```

```
  SimpleEQAudioProcessor::createParameterLayout()
{

}
```

Juce APP class: Generic Interface towards all different audio parameter formats that different plugin hosts use.

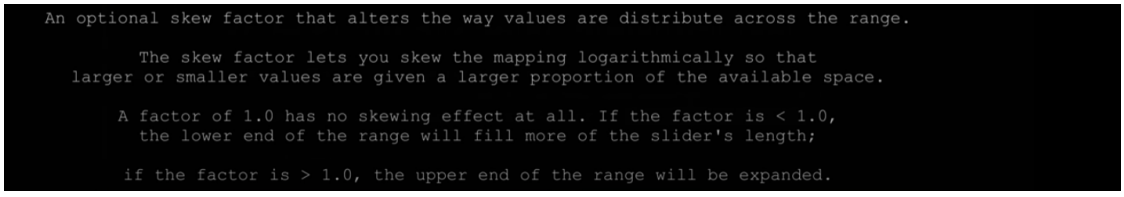
juce class diagram:



AudioParameterFloat: Parameters with a range of values, should be represented with a slider of some kind.

CREATING LOWCUT FREQUENCY PARAMETER:

skew value:




```
juce::AudioProcessorValueTreeState::ParameterLayout SimpleEQAudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;

    layout.add(std::make_unique<juce::AudioParameterFloat>("LowCut Freq", // Name
    "LowCutFreq", // Parameter Name
    juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), //Normalisable Range
    20.f)); // Starting Point


    return layout;
}
```

CREATING HIGH CUT AND PEAK FREQUENCY PARAMETERS:

```
layout.add(std::make_unique<juce::AudioParameterFloat>("HighCut Freq",
"HighCut Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 20000.f));

layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Freq",
"Peak Freq", juce::NormalisableRange<float>(20.f, 20000.f, 1.f, 1.f), 750.f));
```

PEAK GAIN:

parameter expressed in dB: range > -24-24

step size: 0.5 > 0.5 dB change per slider step

slider to behave in linear fashion > skew of 1.f

dont add gain/cut by default > set value of 0

```
layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Gain",
"Peak Gain", juce::NormalisableRange<float>(-24.f, 24.f, 0.5f, 1.f), 0.0f));
```

QUALITY CONTROL OF PEAK (Q)

```
layout.add(std::make_unique<juce::AudioParameterFloat>("Peak Quality",
"Peak Quality", juce::NormalisableRange<float>(0.1f, 10.f, 0.05f, 1.f), 1.f));
```

LOW CUT AND HIGH CUT FILTERS:

Ability to change the steepness of the filter cut

FILTER RESPONSES EXPRESSED IN dB PER OCTAVE.

the way the math behind the filter equations works, it ends up expressing these choices in multiples of 6 or 12:

12, 24, 36, 48 > Specific choices and not a range of values, therefore AudioParameterChoice object is used.

```
//String array that represents choices for filter APC object

    juce::StringArray stringArray;
    for (int i = 0; i < 4; i++)
    {
        juce::String str;
        str << (12 + i * 12);
        str << " dB/Oct";
        stringArray.add(str);
    }
```

CREATING LOW CUT AND HIGH CUT FILTERS AS APC OBJECTS

```
// Low Cut Filter Slope
layout.add(std::make_unique<juce::AudioParameterChoice>("LowCut Slope",
"LowCut Slope", stringArray, 0));
// High Cut Filter Slope
```

```
layout.add(std::make_unique<juce::AudioParameterChoice>("HighCut Slope",
"HighCut Slope", stringArray, 0))
```