## The processor methods

The constructor and destructor are called when the processor is first created and destroyed, respectively.

* NewProjectAudioProcessor ();

Constructor: start off assuming mono input, mono output

* ~NewProjectAudioProcessor ()

A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed by a call to delete.

### How the plugin describes itself

Next, we have several methods that describe how the plugin describes itself to the host

* const String getName() const override { return JucePlugin\_Name; }
* bool acceptsMidi() const override { return false; }
* bool producesMidi() const override { return false; }
* double getTailLengthSeconds() const override { return 0.0; }

Returns the length of the processor's tail, in seconds. The tail length is intended for plugins like delays or reverbs which have an actual tail but not necessarily a look-ahead or other latency.

* bool isBusesLayoutSupported(const BusesLayout& layouts) const override;

// This is the place where you check if the layout is supported. In this template code we only support mono or stereo. Some plugin hosts, such as certain GarageBand versions, will only load plugins that support stereo bus layouts.

… This checks if the input layout matches the output layout

### Setting up the editor

* bool hasEditor() const override { return true; }

Instantiate this plugin's editor/GUI. You can have it return false if you choose to not supply an editor.

* AudioProcessorEditor\* createEditor() override;

### Multiple simultaneously-loaded presets, i.e.programs

int getNumPrograms() override { return 1; }

Some hosts require that ther ise at least 1 program, even if they are not really implemented.

int getCurrentProgram() override { return 0; }

void setCurrentProgram(int) override {}

const String getProgramName(int) override { return {}; }

void changeProgramName(int, const String&) override {}

### Actual audio processing

* void prepareToPlay (double sampleRate, int maxSamplesPerBlock) override;

This method is called at least once before processBlock, to let the processor prepare itself. It is the place to do any pre-playback initialisation that you need.

The sample rate is the target sample rate, and will remain constant until playback stops. The maximumExpectedSamplesPerBlock value is a strong hint about the maximum number of samples that will be provided in each block. The actual block sizes may depend on the host and may be different each time the callback happens.

You can call getTotalNumInputChannels and getTotalNumOutputChannels or query the busLayout member variable to find out the number of channels your processBlock callback must process.

* void releaseResources() override;

When playback stops (audio processing finished), you can use this as an opportunity to free up any allocated memory.

* void processBlock (AudioBuffer<float>&, MidiBuffer&) override;

This method will process one buffer ("block") of data

The following line is included. 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).

It helps 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());

Then it has 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.

### Enabling the host to save and restore state

* void getStateInformation (MemoryBlock& destData) override;

You should use this method to store your parameters in the memory block. You could do that either as raw data, or use the XML or ValueTree classes as intermediaries to make it easy to save and load complex data. The method is called by the host when it needs to persist the current plugin state

* void setStateInformation (const void\* data, int sizeInBytes) override;

You should use this method to restore your parameters from this memory block, whose contents will have been created by the getStateInformation() call. The method is called by the host before processing, when it needs to restore a saved plugin state

### Instantiate this plugin

* AudioProcessor\* JUCE\_CALLTYPE createPluginFilter() { return new NewProjectAudioProcessor (); }

This creates new instances of the plug-in. Note that this is in pluginProcessor.cpp, but it isn’t part of the NewProjectAudioProcessor class

## The editor methods

* NewProjectAudioProcessorEditor::NewProjectAudioProcessorEditor

Make sure that before the constructor has finished, you've set the editor's size to whatever you need it to be.

setSize (400, 300);

* NewProjectAudioProcessorEditor::~NewProjectAudioProcessorEditor()
* void NewProjectAudioProcessorEditor::paint (juce::Graphics& g)

(Our component is opaque, so we must completely fill the background with a solid colour)

* void NewProjectAudioProcessorEditor::resized()

This is generally where you'll want to lay out the positions of any subcomponents in your editor..