Build Your First Audio Plug-in with JUCE

19/11/2021

The JUCE Team -
Attila Szarvas, Ed Davies, Reuben Thomas, Tom Poole

Overview

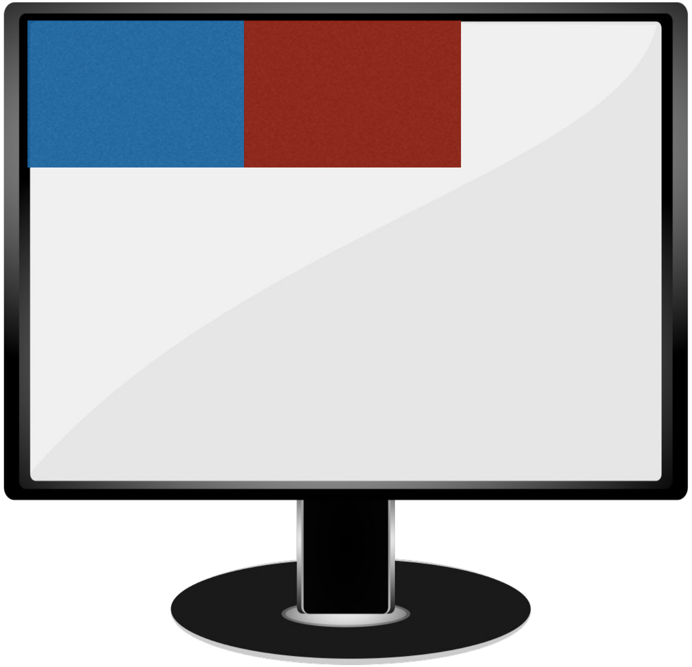
What is JUCE?

Creating JUCE-based projects

Building audio plug-ins

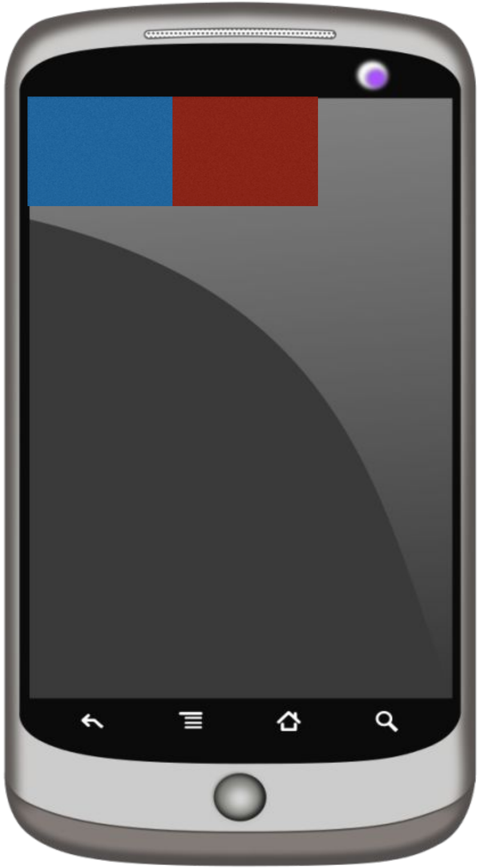
Testing audio plug-ins

```
{
    setColour (blue);
    drawRect (0, 0, 100, 50);

    setColour (red);
    drawRect (100, 0, 100, 50);
}
```

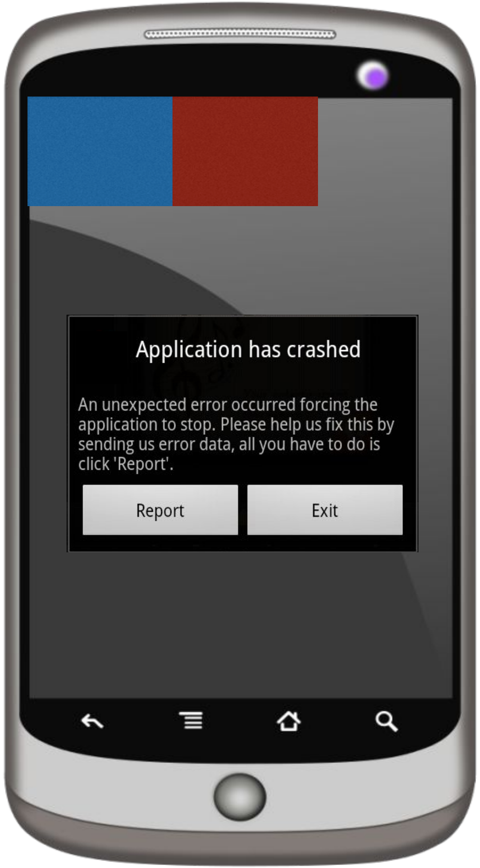


```
{
        setColour (blue);
        drawRect (0, 0, 100, 50);

        setColour (red);
        drawRect (100, 0, 100, 50);
}
```

```
{
        setColour (blue);
        drawRect (0, 0, 100, 50);

        setColour (red);
        drawRect (100, 0, 100, 50);
}
```



```
{
        setColour (blue);
        drawRect (0, 0, 100, 50);

        setColour (red);
        drawRect (100, 0, 100, 50);
}
```

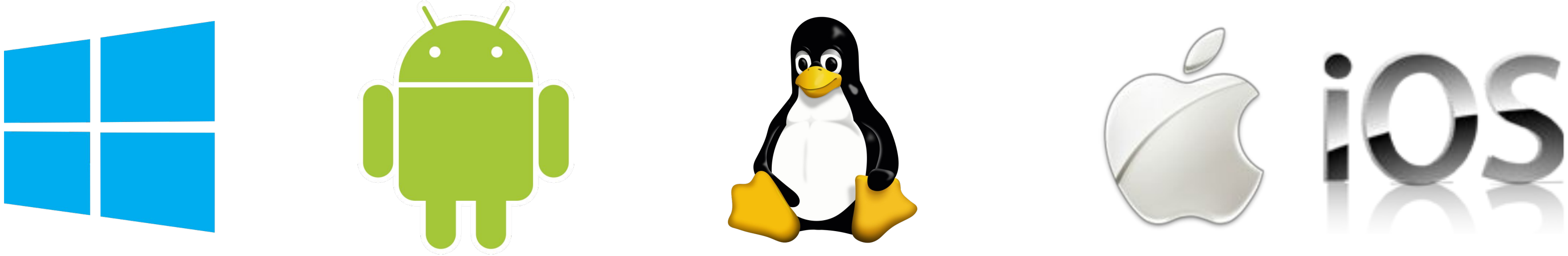


Application has crashed

An unexpected error occurred forcing the application to stop. Please help us fix this by sending us error data, all you have to do is click 'Report'.

```
{
        setColour (blue);
        drawRect (0, 0, 100, 50);

        setColour (red);
        drawRect (100, 0, 100, 50);
}
```

C++ C++ C++ Obj-C++



Win32

Native Activity

POSIX/X11

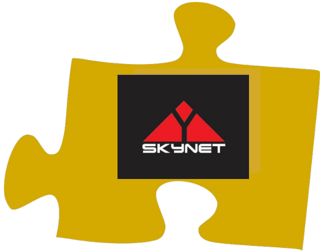
AppKit/UIKit

macOS/iOS

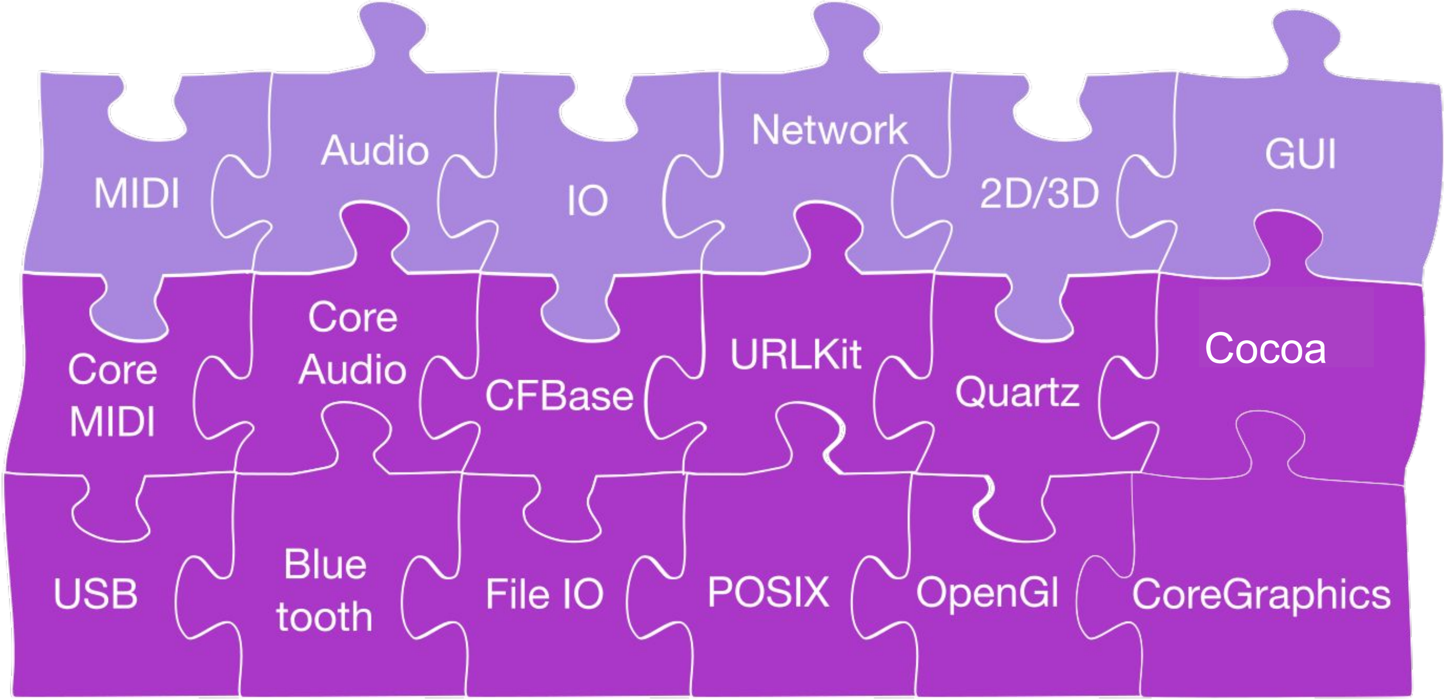
```
{
    CGRect rc = { 0, 0, 100, 50 };
    CGContextSetRGBFillColor (context, 0.0f, 0.0f, 1.0f);
    CGContextFillRect (context, &rc);

    rc = { 100, 0, 100, 50 };
    CGContextSetRGBFillColor (context, 1.0f, 0.0f, 0.0f);
    CGContextFillRect (context, &rc);
}
```

Windows

```
{
    Rect rc = { 0, 0, 100, 50 };
    FillRect (context, &rc, RGB (0, 0, 255));

    rc = { 100, 0, 100, 50};
    FillRect (context, &rc, RGB (255, 0, 0));
}
```


SKYNET

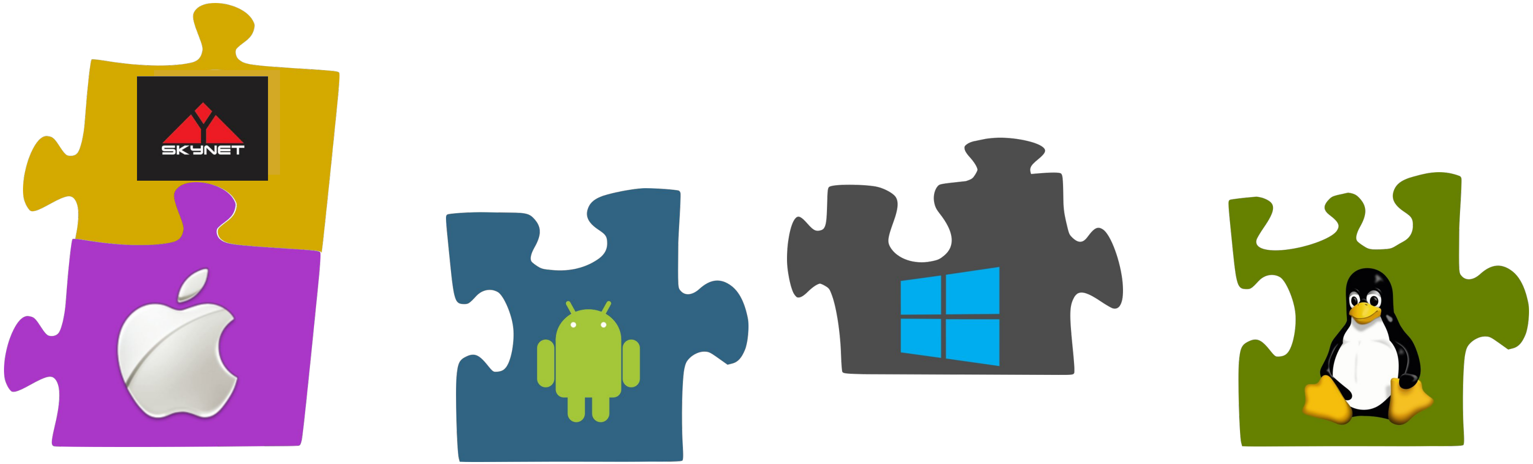
SKYNET

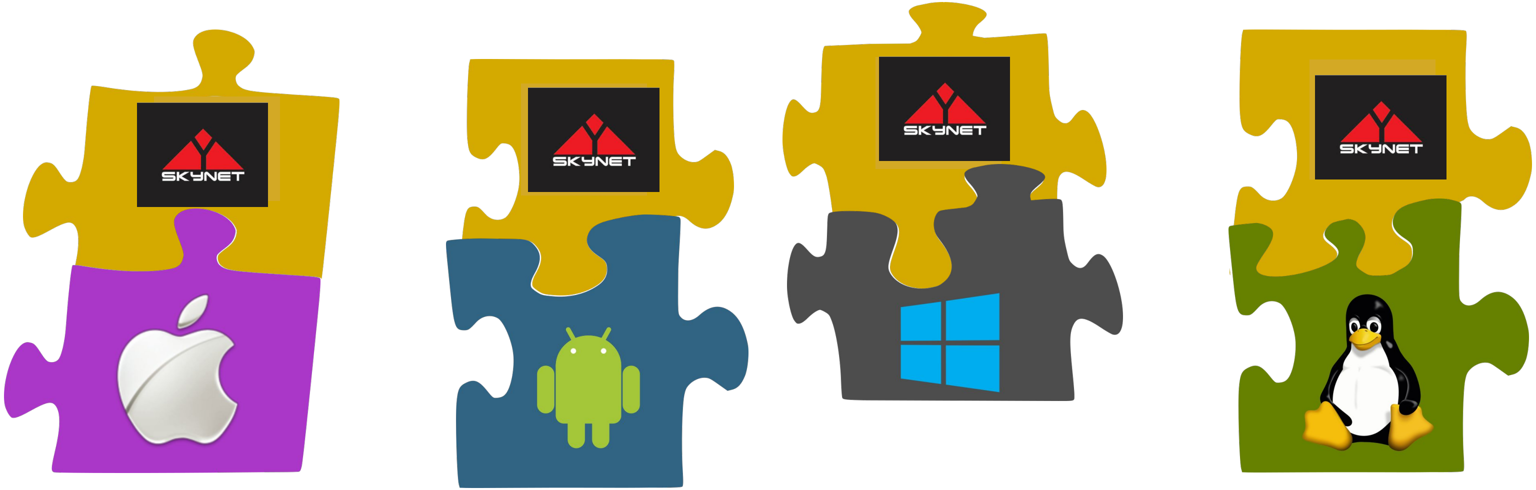


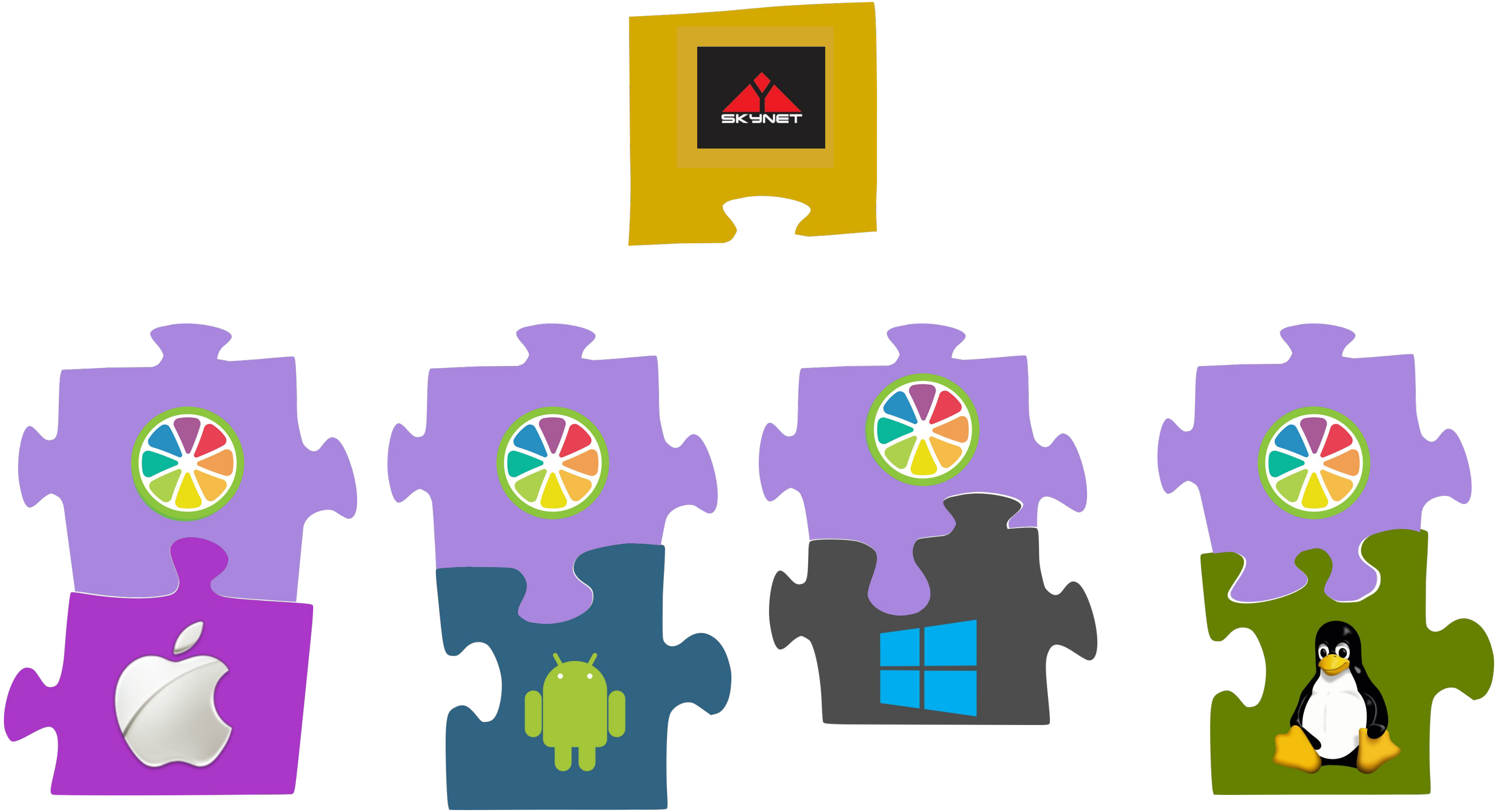
MIDI Audio IO Network 2D/3D GUI

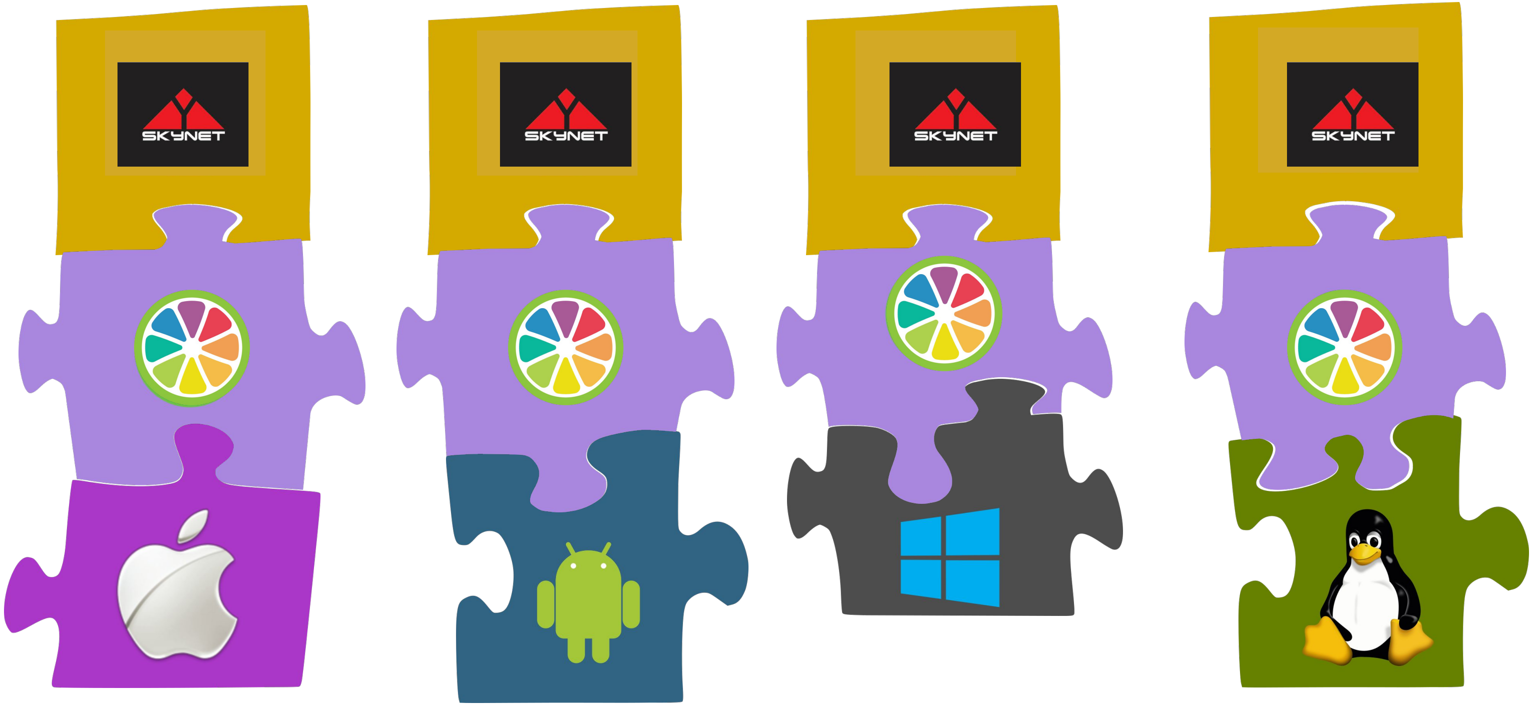
Core MIDI Core Audio CFBase URLKit Quartz Cocoa

USB Bluetooth File IO POSIX OpenGl CoreGraphics

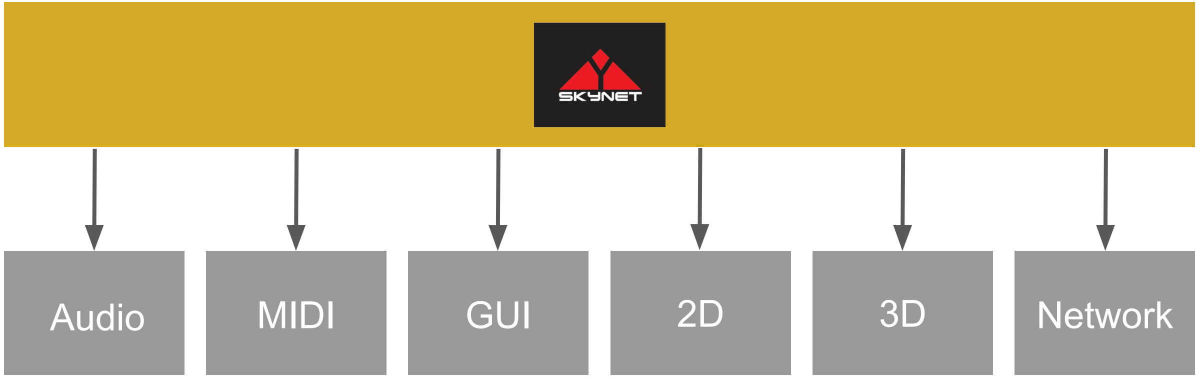

SKYNET

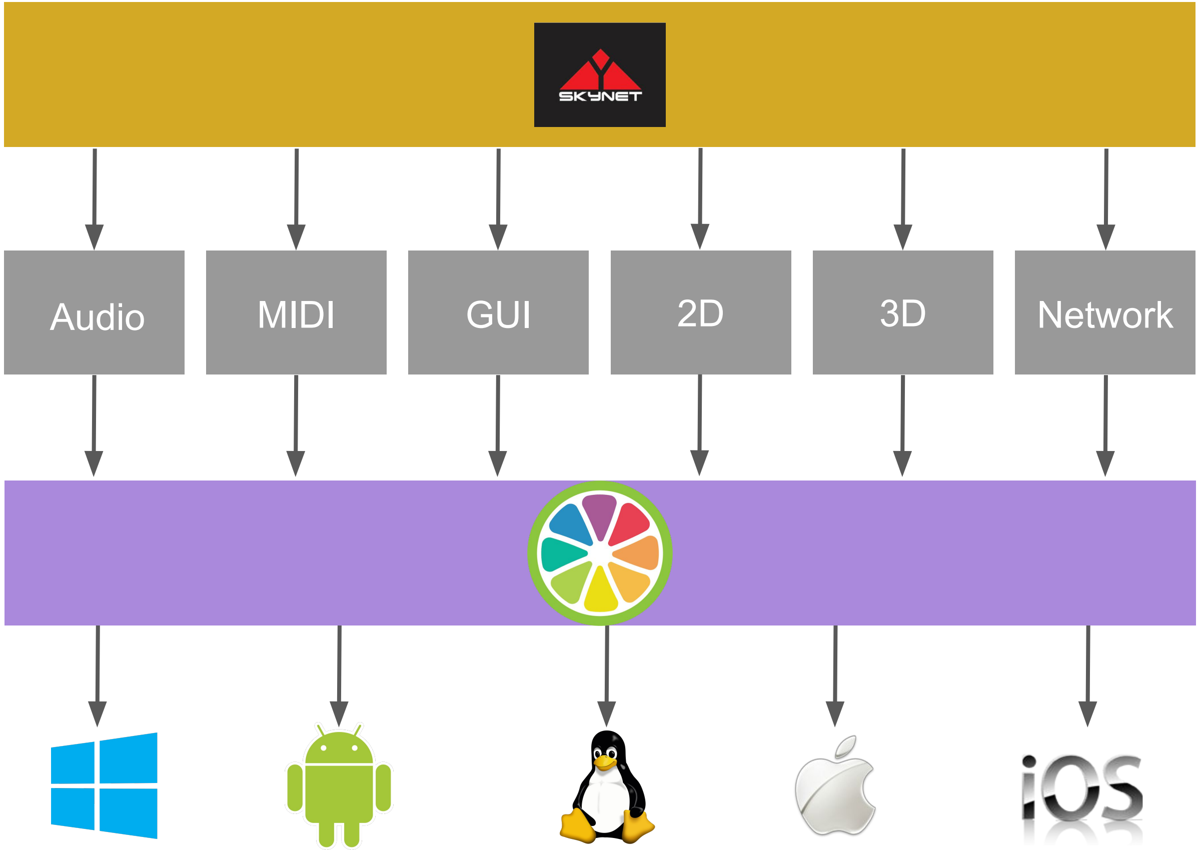

SKYNET
SKYNET
SKYNET
SKYNET


SKYNET

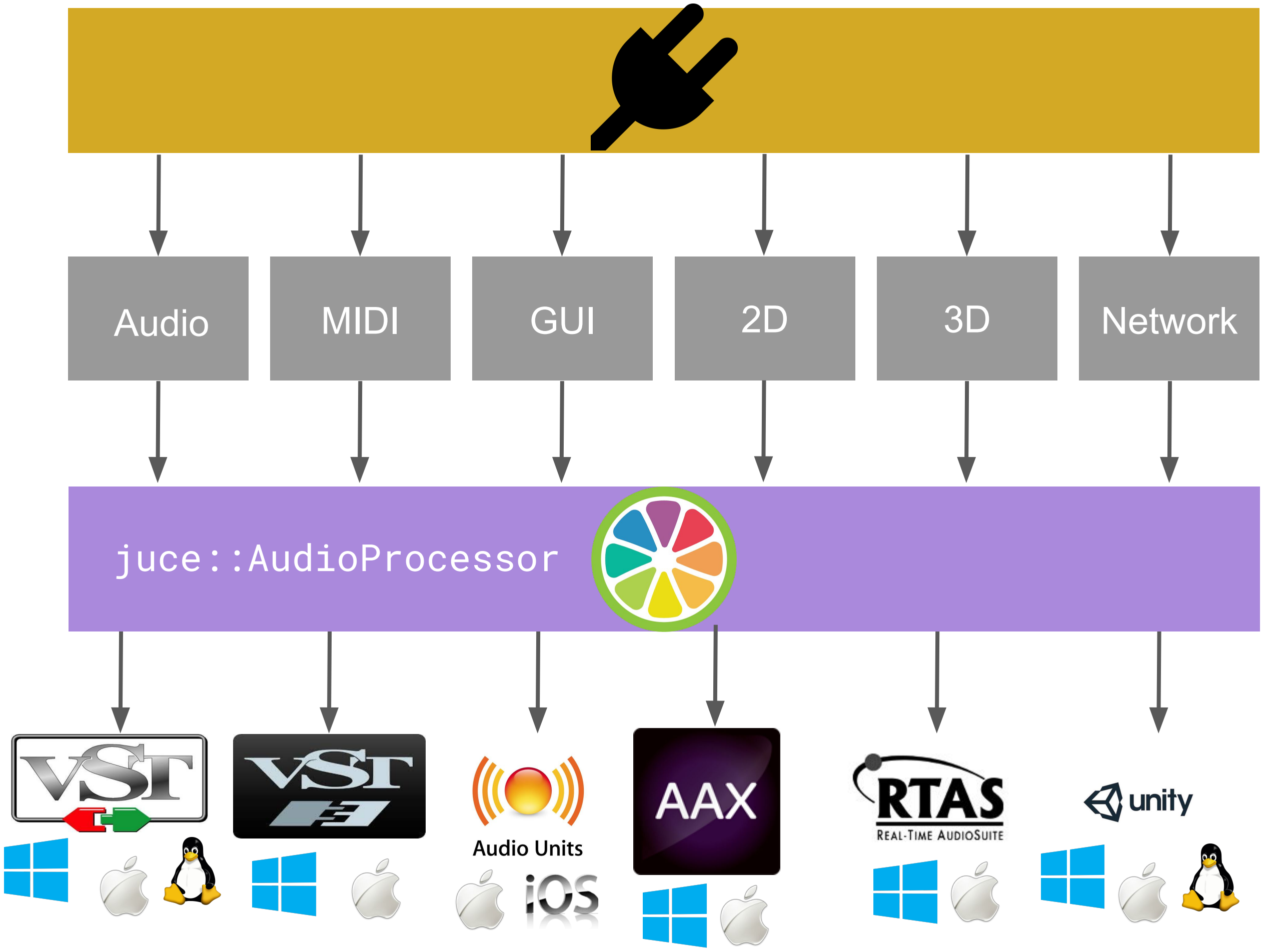

SKYNET
SKYNET
SKYNET
SKYNET

```
{
    g.setColour (juce::Colours::blue);
    g.drawRect (0, 0, 100, 50);


    g.setColour (juce::Colours::red);
    g.drawRect (100, 0, 100, 50);
}
```


SKYNET
Audio
MIDI
GUI
2D
3D
Network



SKYNET
Audio
MIDI
GUI
2D
3D
Network
iOS



Audio

MIDI

GUI

2D

3D

Network

`juce::AudioProcessor`

Building an Audio Plug-in

During the workshop

- All the slides are numbered
- Post questions in the Discord channel
- We’ll take a couple of breaks during the session
- The slides are available in the workshop materials, so you can revisit sections

Creating a plug-in using JUCE

We're going to walk through the creation of a simple delay effect plug-in

- Setting up a JUCE project using the Projucer
- Writing the C++ plug-in code in an IDE
- Compiling the different plug-in formats
- Adding plug-in parameters
- Creating a GUI
- Testing the plug-in

What won’t be covered
(but you may want to look into)

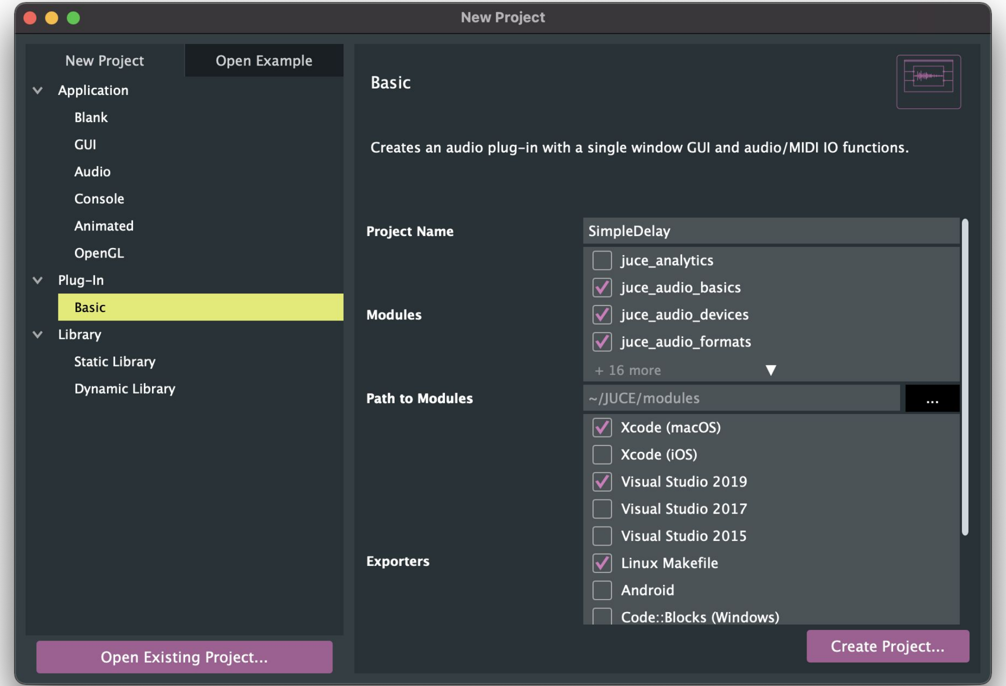
- Plug-in configurations - multibus, surround
- Mobile platforms
- Performance optimisations
  - Profiling
  - Compiler settings
- Release process
  - Installers
  - Code signing

Workshop materials

Contents:

- The `workspace` directory is where we'll build our plug-in
- The `Plugins` directory is where you will find your compiled plug-ins
- Numbered directories containing source code snapshots corresponding to each section

Creating a plug-in project



Basic

Creates an audio plug-in with a single window GUI and audio/MIDI IO functions.

Project Name: SimpleDelay

Modules
- ☐ juce_analytics
- ☑ juce_audio_basics
- ☑ juce_audio_devices
- ☑ juce_audio_formats
- + 16 more ▼

Path to Modules: ~/JUCE/modules ...

Exporters
- ☑ Xcode (macOS)
- ☐ Xcode (iOS)
- ☑ Visual Studio 2019
- ☐ Visual Studio 2017
- ☐ Visual Studio 2015
- ☑ Linux Makefile
- ☐ Android
- ☐ Code::Blocks (Windows)

#01: Creating a plug-in project

We're now using the workshop checkpoints

- The #01 in the title signals that we're using the contents of the `01` directory
- To return to this point in the workshop you can delete the contents of the `workspace` folder and copy the contents of the `01` folder into the `workspace` folder
- We've already done this for step 01, so we're all starting from the same place

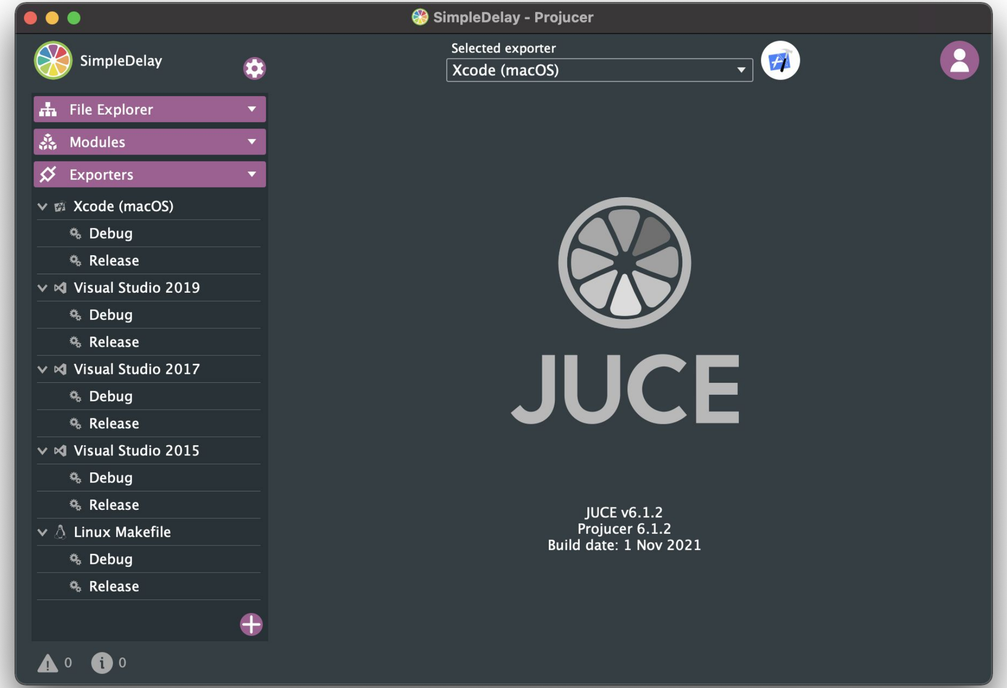
#01: Creating a plug-in project

Objectives of this section:

- Go through the common project configurations settings in the Projucer
- Export the project and open it in your IDE of choice
    - Xcode
    - Visual Studio
    - Whichever Linux editor you want to use
- Build the empty template project
- Load the plug-in in the TestHost

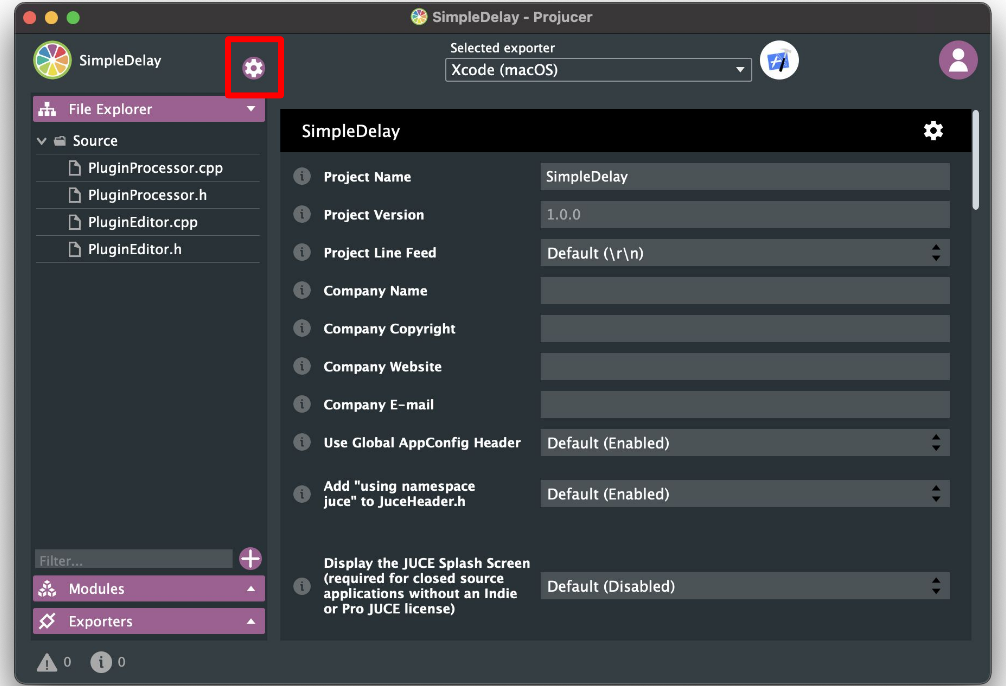
In the Projucer, open the `workspace/SimpleDelay.jucer` project.

#01: The project



JUCE v6.1.2
Projucer 6.1.2
Build date: 1 Nov 2021

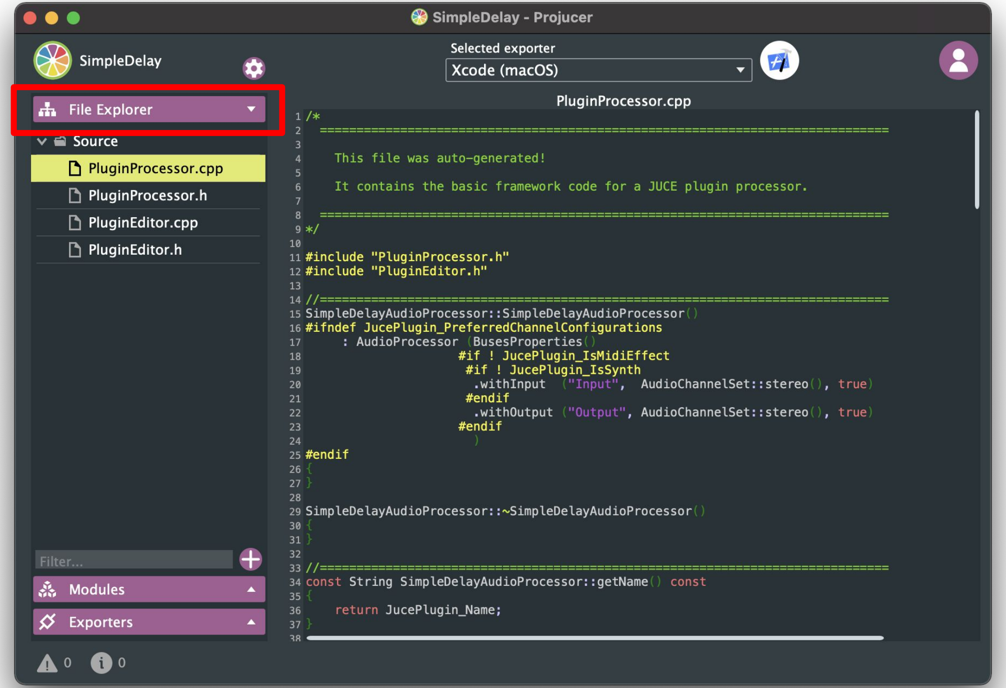
#01: Project settings



SimpleDelay

Project Name: SimpleDelay
Project Version: 1.0.0
Project Line Feed: Default (\r\n)
Company Name:
Company Copyright:
Company Website:
Company E-mail:
Use Global AppConfig Header: Default (Enabled)
Add "using namespace juce" to JuceHeader.h: Default (Enabled)
Display the JUCE Splash Screen (required for closed source applications without an Indie or Pro JUCE license): Default (Disabled)

#01: File explorer

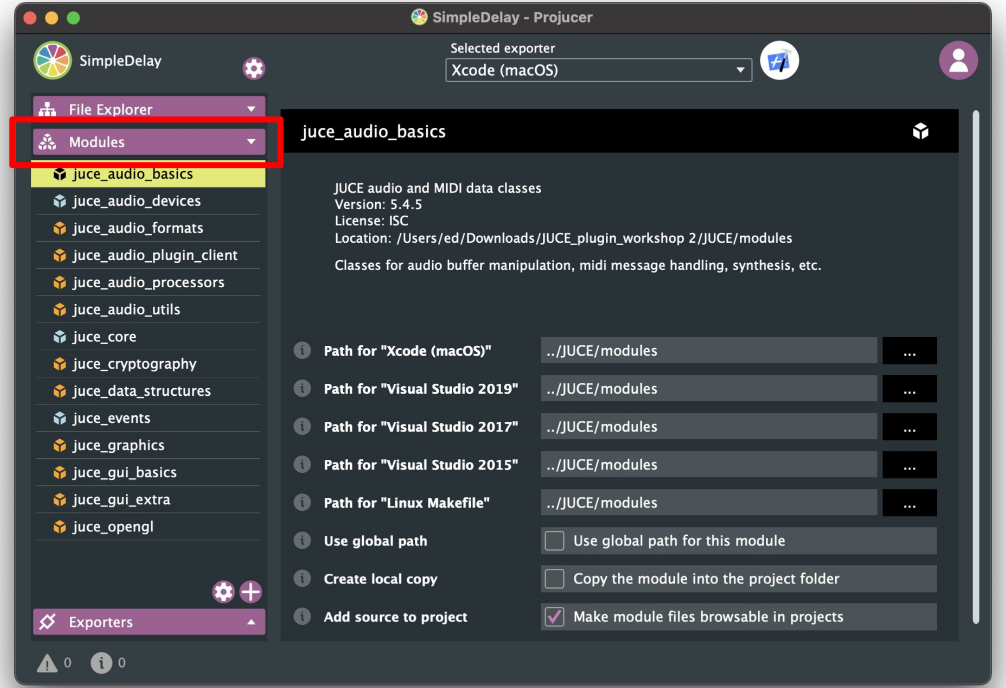


Selected exporter: Xcode (macOS)

PluginProcessor.cpp

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

    This file was auto-generated!

    It contains the basic framework code for a JUCE plugin processor.

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

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
}

SimpleDelayAudioProcessor::~SimpleDelayAudioProcessor()
{
}

//==============================================================================
const String SimpleDelayAudioProcessor::getName() const
{
    return JucePlugin_Name;
}
```

#01: Module settings



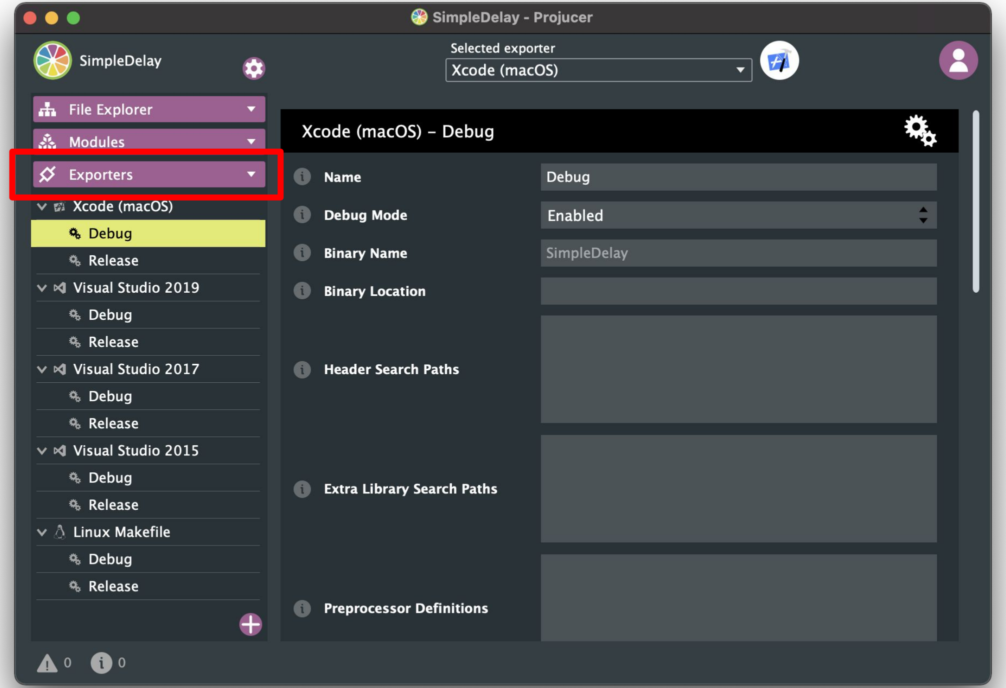
juce_audio_basics

JUCE audio and MIDI data classes
Version: 5.4.5
License: ISC
Location: /Users/ed/Downloads/JUCE_plugin_workshop 2/JUCE/modules

Classes for audio buffer manipulation, midi message handling, synthesis, etc.

Path for "Xcode (macOS)"	../JUCE/modules ...
Path for "Visual Studio 2019"	../JUCE/modules ...
Path for "Visual Studio 2017"	../JUCE/modules ...
Path for "Visual Studio 2015"	../JUCE/modules ...
Path for "Linux Makefile"	../JUCE/modules ...
Use global path	☐ Use global path for this module
Create local copy	☐ Copy the module into the project folder
Add source to project	☑ Make module files browsable in projects

#01: Exporter Settings

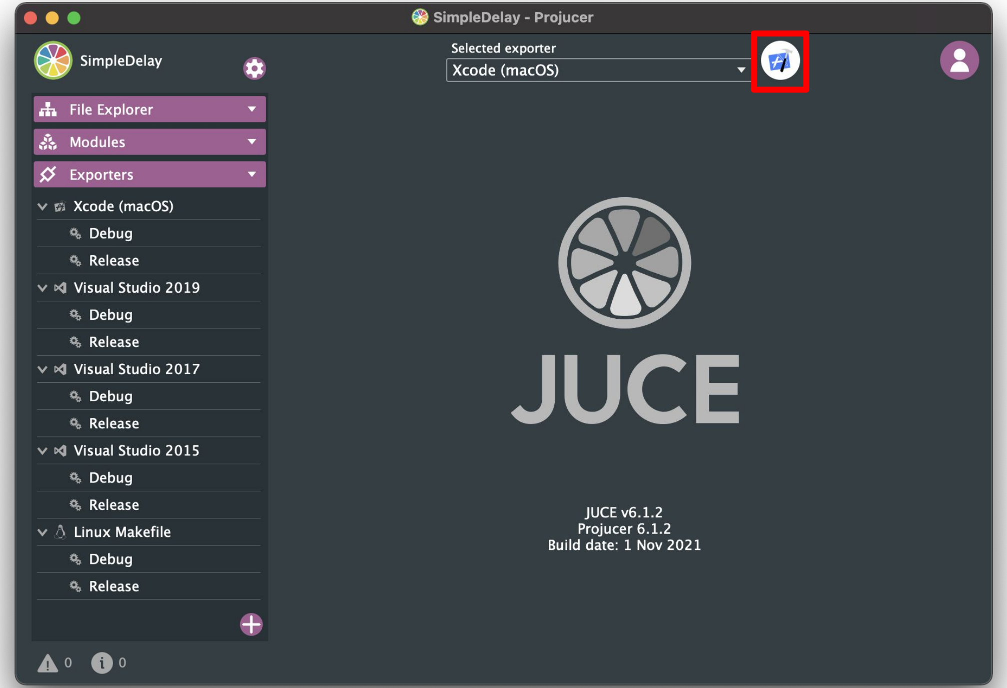


Selected exporter: Xcode (macOS)

Xcode (macOS) – Debug

Name	Debug
Debug Mode	Enabled
Binary Name	SimpleDelay
Binary Location	
Header Search Paths	
Extra Library Search Paths	
Preprocessor Definitions	

#01: Generate IDE project files



SimpleDelay - Projucer

SimpleDelay

Selected exporter: Xcode (macOS)

- Xcode (macOS)
  - Debug
  - Release
- Visual Studio 2019
  - Debug
  - Release
- Visual Studio 2017
  - Debug
  - Release
- Visual Studio 2015
  - Debug
  - Release
- Linux Makefile
  - Debug
  - Release

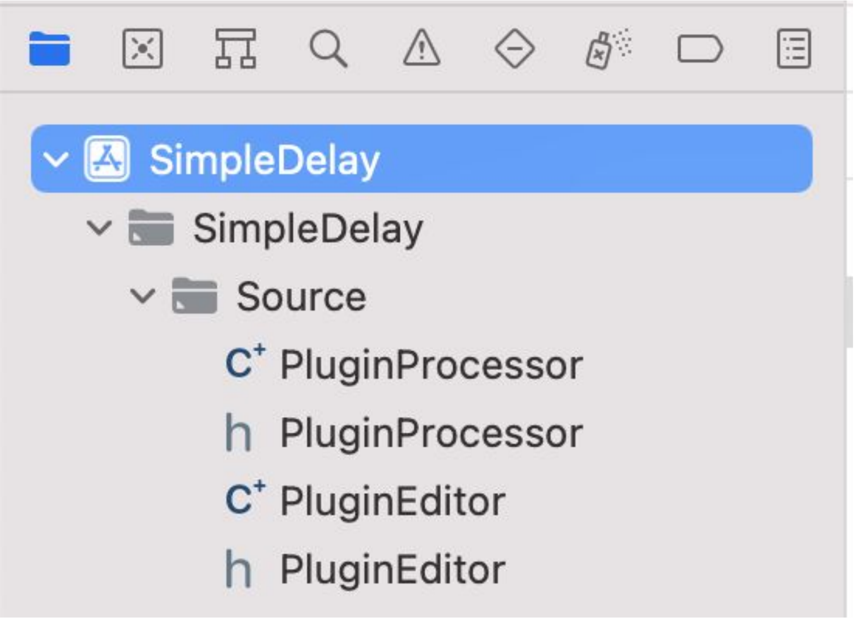
JUCE v6.1.2
Projucer 6.1.2
Build date: 1 Nov 2021

#01: Generate IDE project files

We’ve seen a lot of Projucer settings but we don’t need to change any for this workshop!

Creating a Linux Makefile is a little different - there are no IDEs associated with Makefiles so after saving your project you need to run `make` from the command line

#01: Project structure

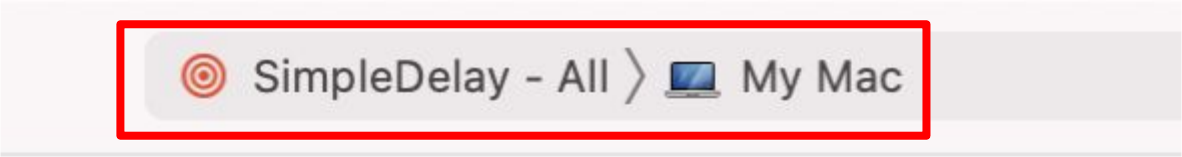


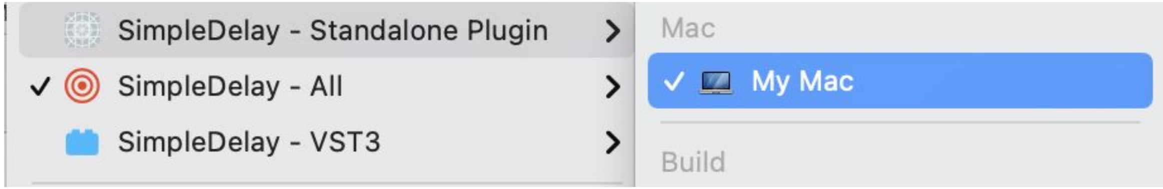
SimpleDelay
SimpleDelay
Source
PluginProcessor
PluginProcessor
PluginEditor
PluginEditor

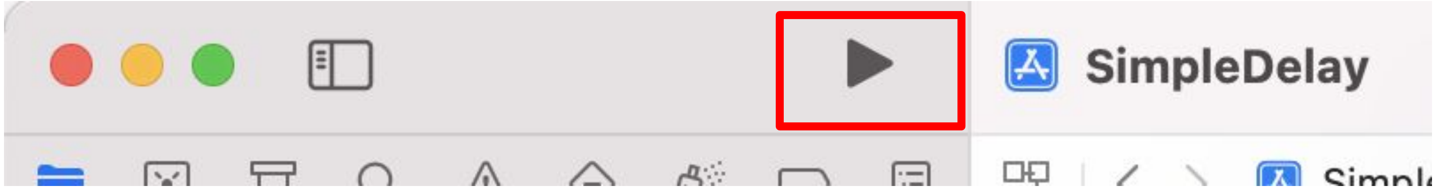
`AudioProcessor`

`AudioProcessorEditor`

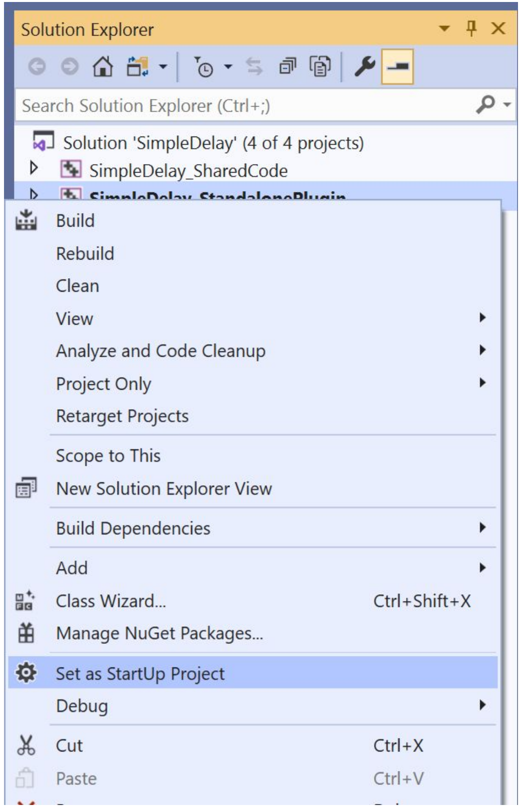
#01: Running Standalone (macOS)

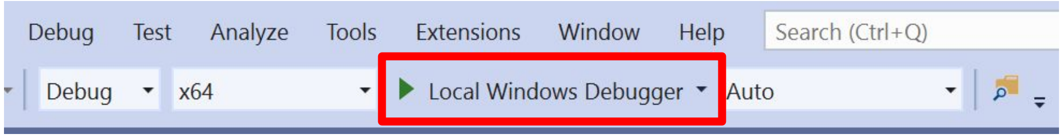






#01: Running Standalone (Windows)

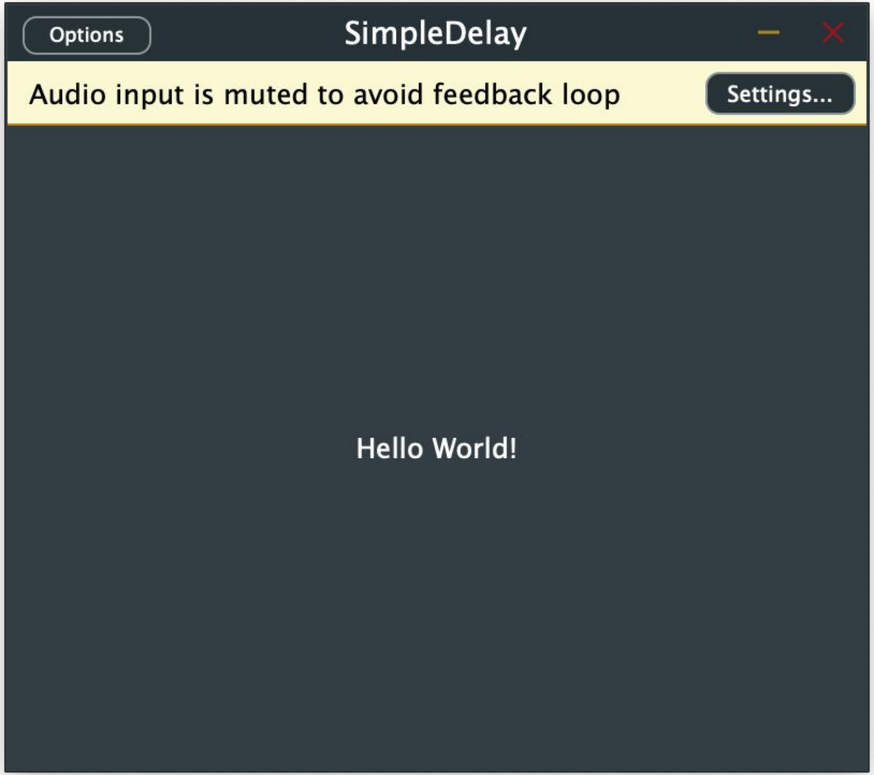




#01: Running Standalone (Linux)

```
cd workspace/Builds/LinuxMakefile

make

./build/SimpleDelay
```

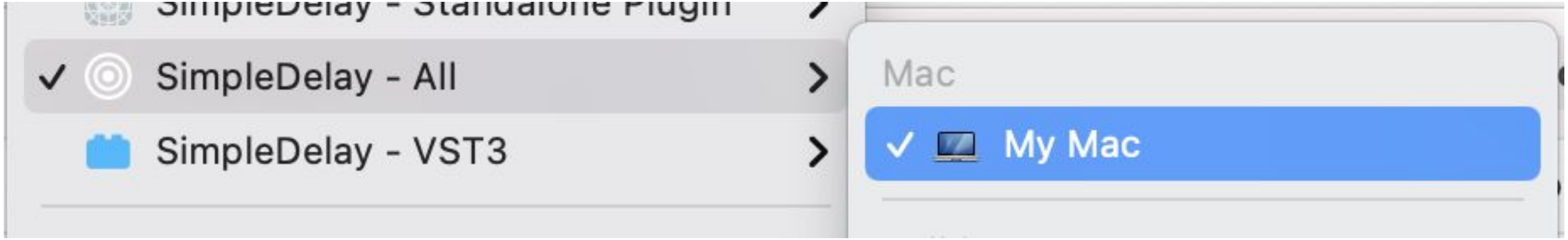
#01: Running Standalone

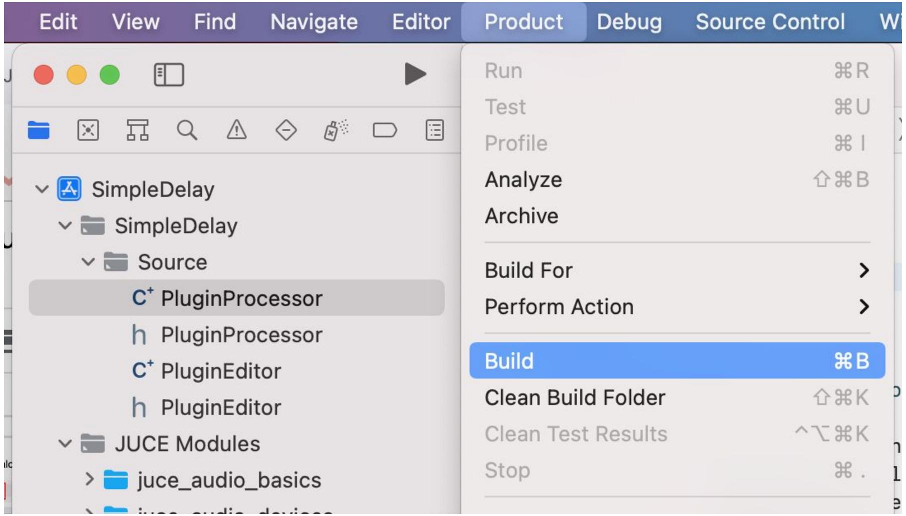


Audio input is muted to avoid feedback loop

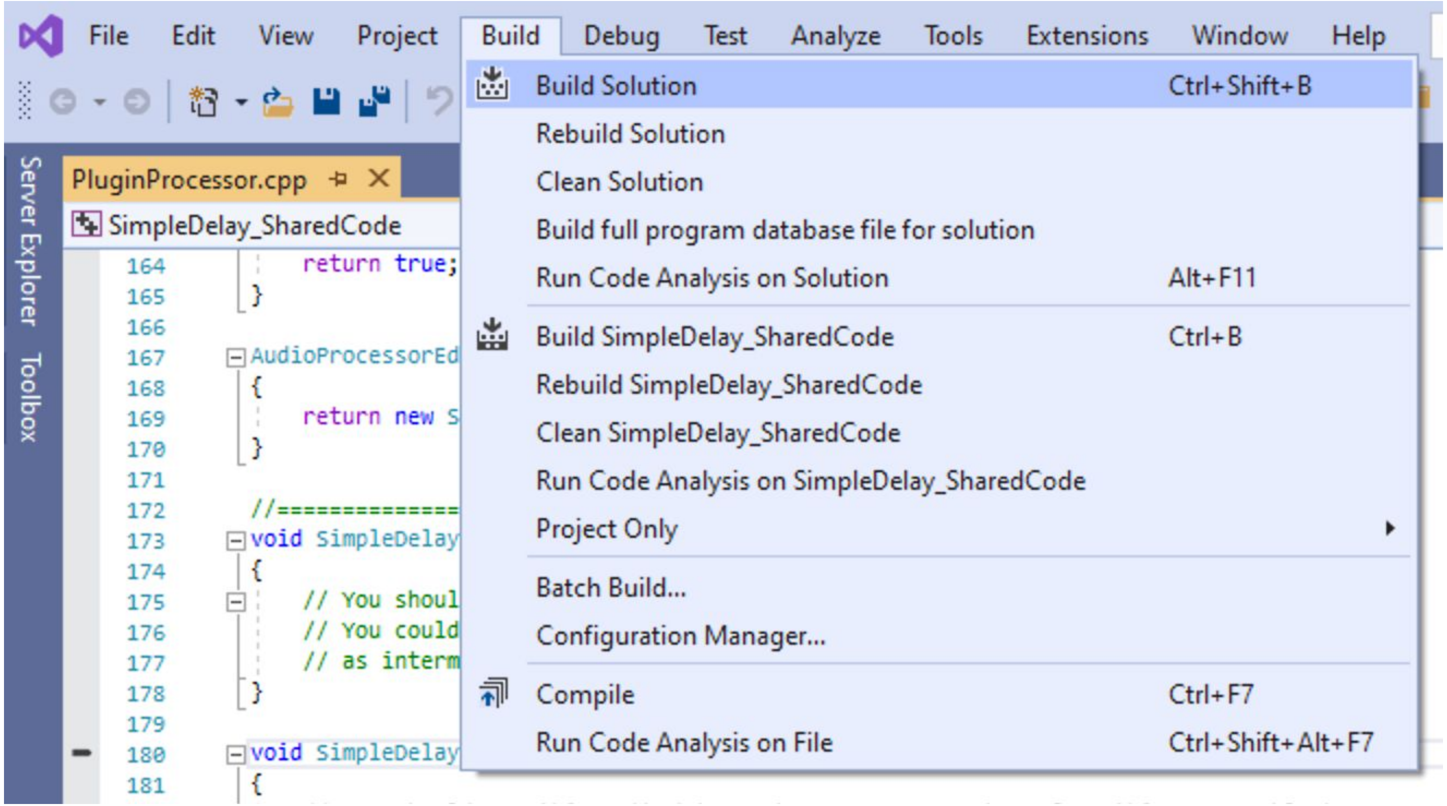
Hello World!

#01: Building plug-ins (macOS)





#01: Building plug-ins (Windows)


File
Edit
View
Project
Build
Debug
Test
Analyze
Tools
Extensions
Window
Help
Server Explorer
Toolbox
PluginProcessor.cpp
SimpleDelay_SharedCode
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Build full program database file for solution
Run Code Analysis on Solution Alt+F11
Build SimpleDelay_SharedCode Ctrl+B
Rebuild SimpleDelay_SharedCode
Clean SimpleDelay_SharedCode
Run Code Analysis on SimpleDelay_SharedCode
Project Only
Batch Build...
Configuration Manager...
Compile Ctrl+F7
Run Code Analysis on File Ctrl+Shift+Alt+F7

```
    return true;
}

AudioProcessorEd
{
    return new S
}

//==============
void SimpleDelay
{
    // You shoul
    // You could
    // as interm
}

void SimpleDelay
{
```

#01: Building plug-ins (Linux)

```
cd workspace/Builds/LinuxMakefile

make
```

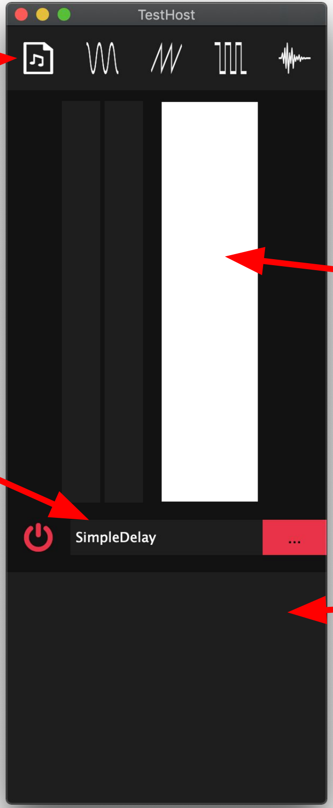
#01: Running in the TestHost

A simple host application for testing our plug-in can be found in the workshop materials:

- `TestHost/macOS/TestHost.app`
- `TestHost/Win64/TestHost.exe`
- `TestHost/Linux/TestHost`

Launch the one appropriate for your platform.

#01: Running in the TestHost



Audio input sources

Gain fader & level meters

Plug-in section

Oscilloscope

#01: Running in the TestHost

Find the SimpleDelay plug-in that you've built in the `Plugins` folder and drop it onto the plug-in section of the TestHost app.

Click on the plug-in name to open the UI.

Break

#02: Modifying audio

Objectives of this section:

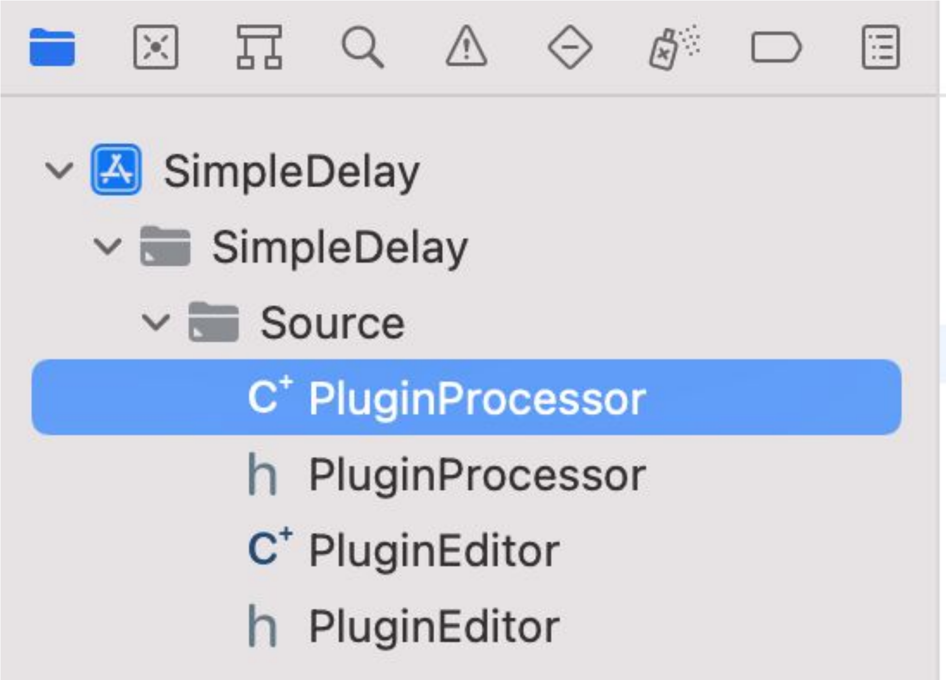
- Edit source files in your IDE
- Add a very simple gain reduction to the plug-in
- Listen to the results

Copy the contents of the `02` directory into `workspace`.

Open your IDE project file

- `workspace/Builds/VisualStudio2019/SimpleDelay.sln`
- `workspace/Builds/macOS/SimpleDelay.xcodeproj`

#02: Edit the `AudioProcessor`



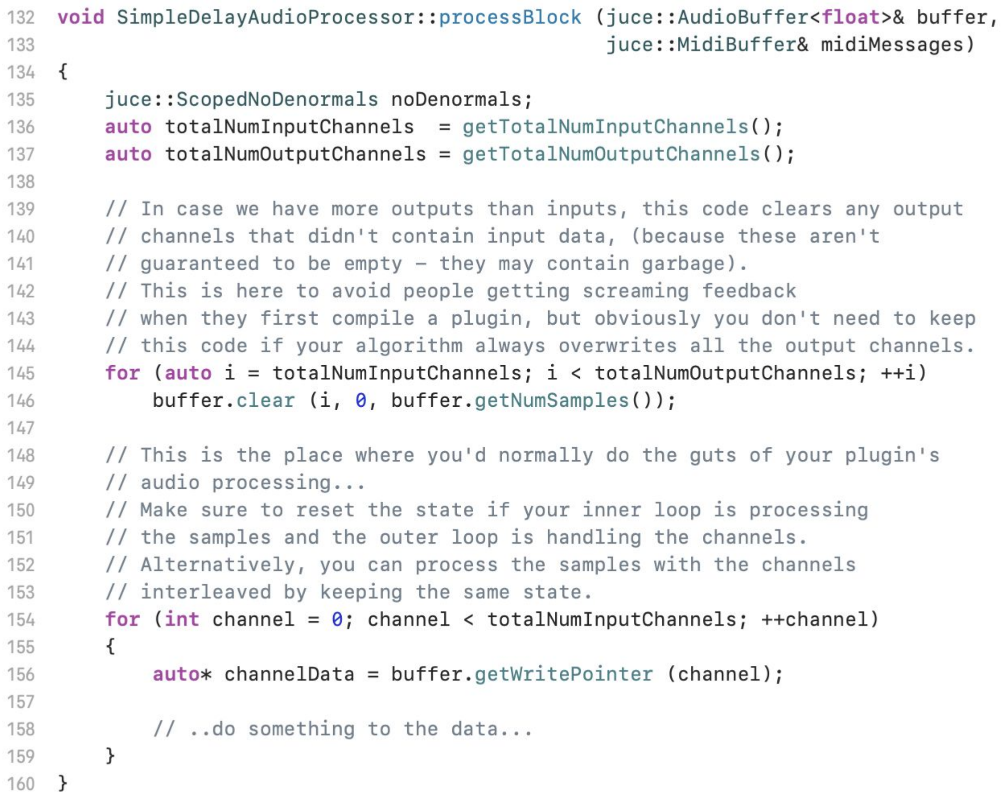
#02: The `SimpleDelayAudioProcessor` class

The `SimpleDelayAudioProcessor` class has a lot of methods

- All of these methods have been automatically written by the template created by the Projucer
- You can change the contents of these methods to change the behavior of the plug-in

The only one needed for this section is `processBlock`

#02: `processBlock`



```
void SimpleDelayAudioProcessor::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...
    }
}
```

#02: What does `processBlock` do?

`processBlock` is called repeatedly by the plug-in host with a chunk of audio to process

For plug-ins the amount of audio to process usually corresponds to the “buffer size” setting of the host (1024, 512, …)

Be careful here! `processBlock` will be called on the *audio* thread, which is not the same as that used for drawing a GUI or handling mouse events (more on this a little later)

#02: `juce::AudioBuffer<float>`

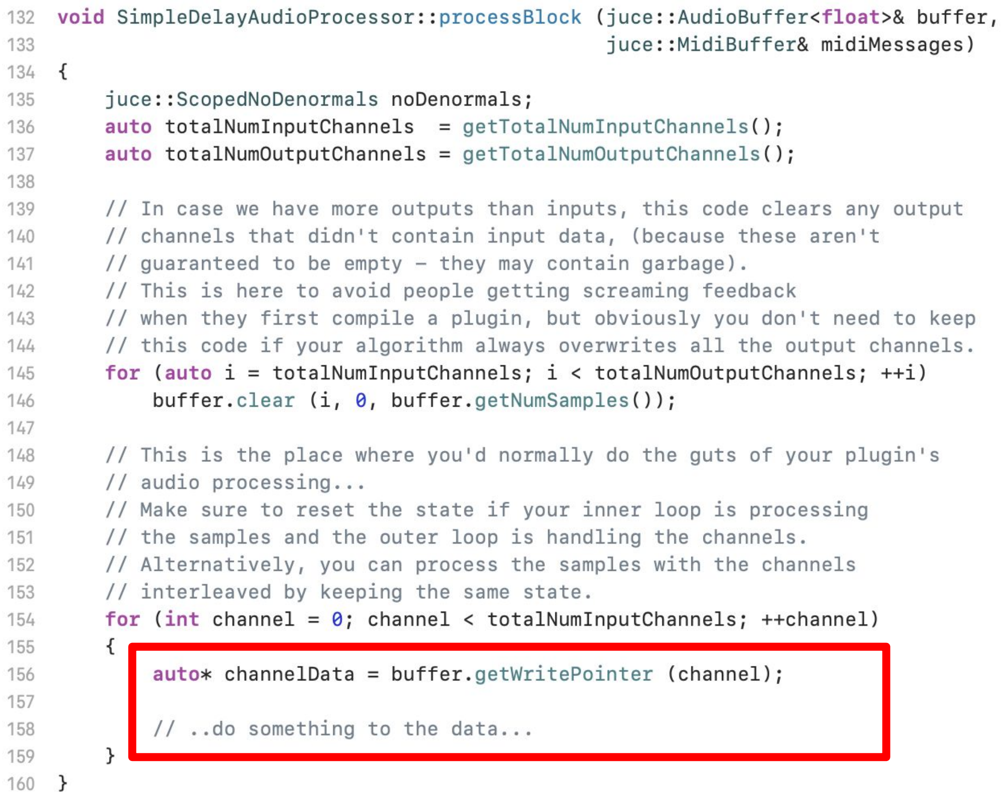
`AudioBuffer` is a class containing non-interleaved channels of audio data represented as floats, and methods to access and modify them

Non-interleaved: separate channels of continuous audio data

Floats: each sample is represented by a floating point number in the range -1.0 to 1.0

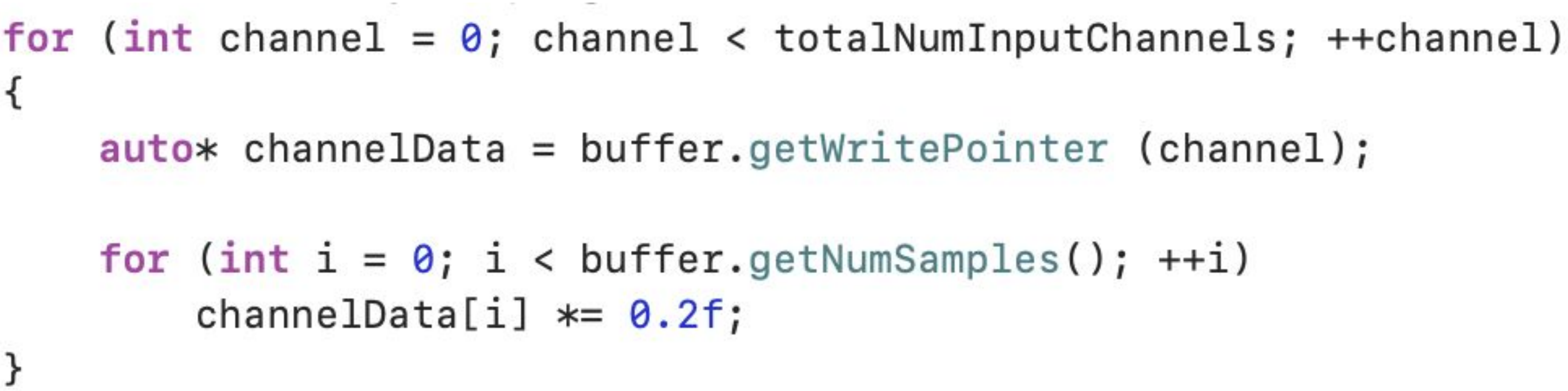
`processBlock` is passed a reference to an `AudioBuffer` containing the incoming audio data. We create an audio effect by modifying this data.

#02: Modifying audio



```
void SimpleDelayAudioProcessor::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...
    }
}
```

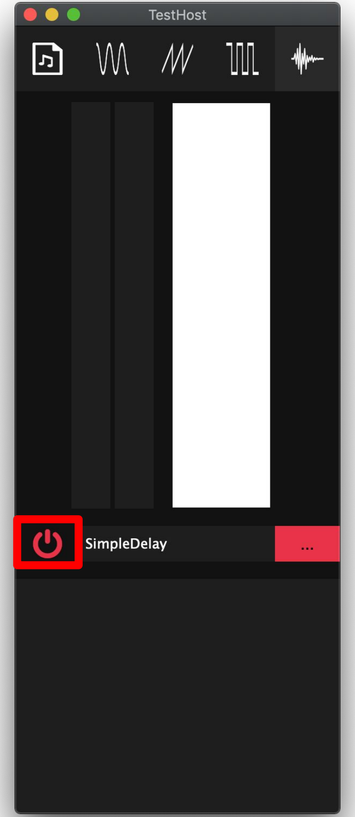
#02: Fixed gain reduction



```
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    auto* channelData = buffer.getWritePointer (channel);

    for (int i = 0; i < buffer.getNumSamples(); ++i)
        channelData[i] *= 0.2f;
}
```

#02: Testing the fixed gain reduction

- Load the plug-in in the TestHost
- Toggle the bypass
- Hear the gain reduction when audio is going through the plug-in



#03: Plug-in parameters

Objectives of this section:

Create a parameter to control the gain reduction

- Add a parameter to your plug-in interface
- Use the parameter the change how the audio is processed
- Change the parameter value in a (auto-generated) user interface
- Listen to how the audio is changed in real time

#03: What is a parameter?

Parameters are how plug-in hosts control plug-ins

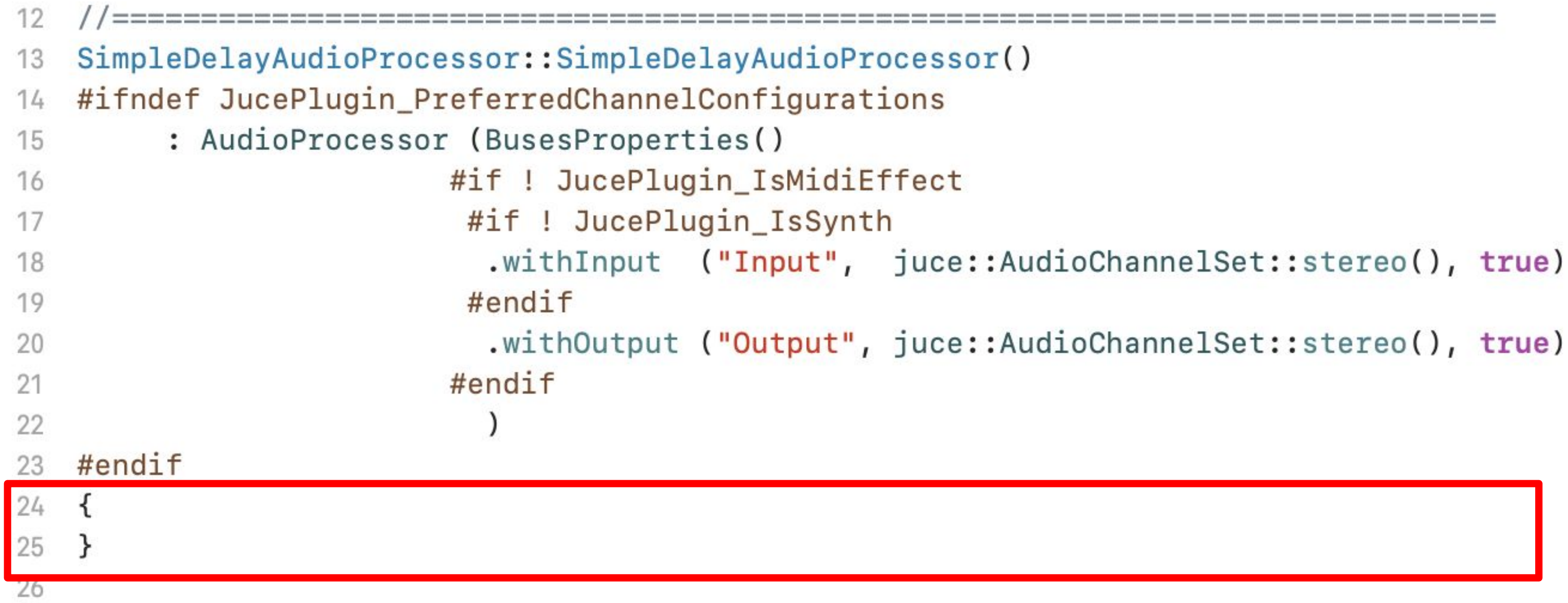
- They are exposed as part of the plug-in format’s interface
  - Hosts can query plug-ins to find out what parameters they offer
- Parameters can be changed via the plug-in’s GUI
- Parameters can be changed via automation
  - Though they can also be marked as non-automatable

#03: Parameter types

All derived from the `AudioProcessorParameter` class

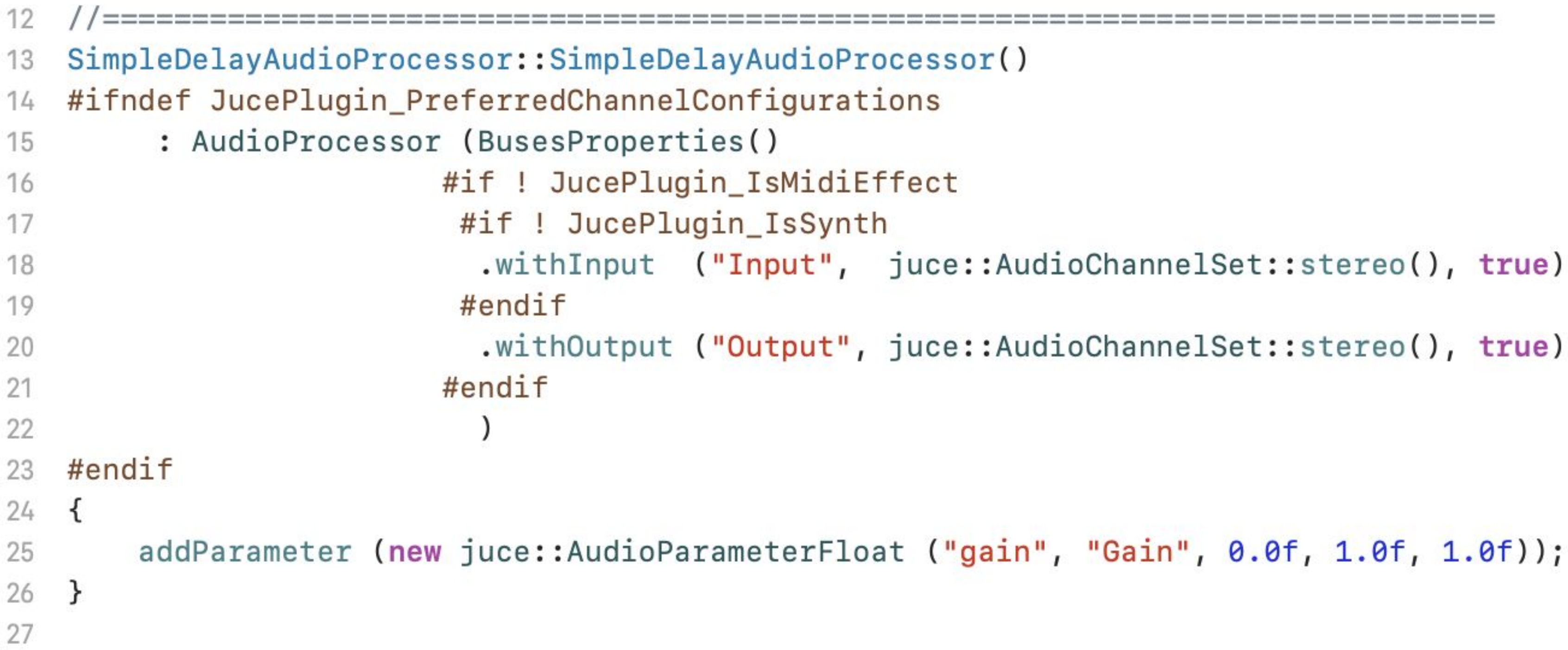
- Methods for getting and setting parameter values and properties
- Added to, and then managed by, your `AudioProcessor`
- JUCE provides some basic types to get your started
  - `AudioParameterFloat`
  - `AudioParameterBool`
  - `AudioParameterChoice`
  - `AudioParameterInt`

#03: Adding a parameter



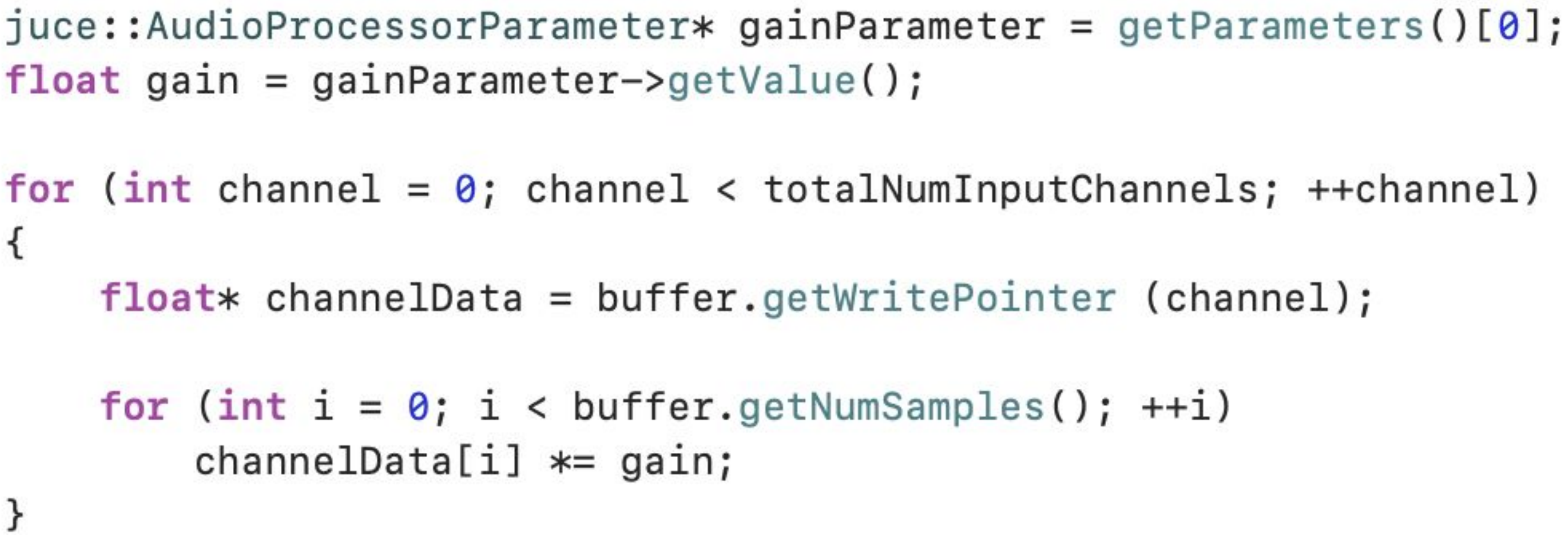
```
//==============================================================================
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
}
```

#03: Adding a parameter



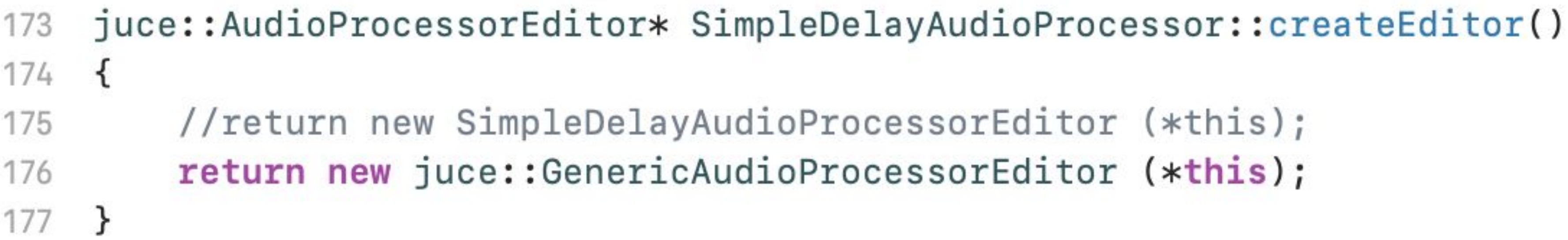
```
//==============================================================================
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
    addParameter (new juce::AudioParameterFloat ("gain", "Gain", 0.0f, 1.0f, 1.0f));
}
```

#03: Using a parameter in `processBlock`



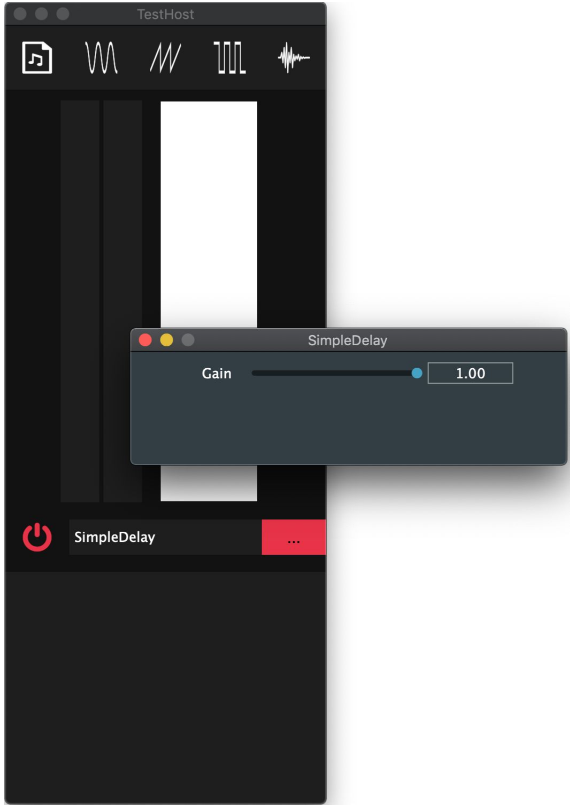
```
juce::AudioProcessorParameter* gainParameter = getParameters()[0];
float gain = gainParameter->getValue();

for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    float* channelData = buffer.getWritePointer (channel);

    for (int i = 0; i < buffer.getNumSamples(); ++i)
        channelData[i] *= gain;
}
```

#03: Use an automatically generated user interface



```
juce::AudioProcessorEditor* SimpleDelayAudioProcessor::createEditor()
{
    //return new SimpleDelayAudioProcessorEditor (*this);
    return new juce::GenericAudioProcessorEditor (*this);
}
```

#03: Compile the plug-in and load it



#04: Creating the delay effect

Objectives of this section:

Add some less trivial audio processing

- Add some more parameters
- Configure our audio processing algorithm in `prepareToPlay`
- Implement a basic delay effect
- Play with the effect in TestHost

This section requires quite a lot of typing to complete. Start from the contents of the `05` folder and review the changes rather than typing along.

Begin compiling the plug-in formats now!

#04: What is a delay effect?

An echo of the incoming audio

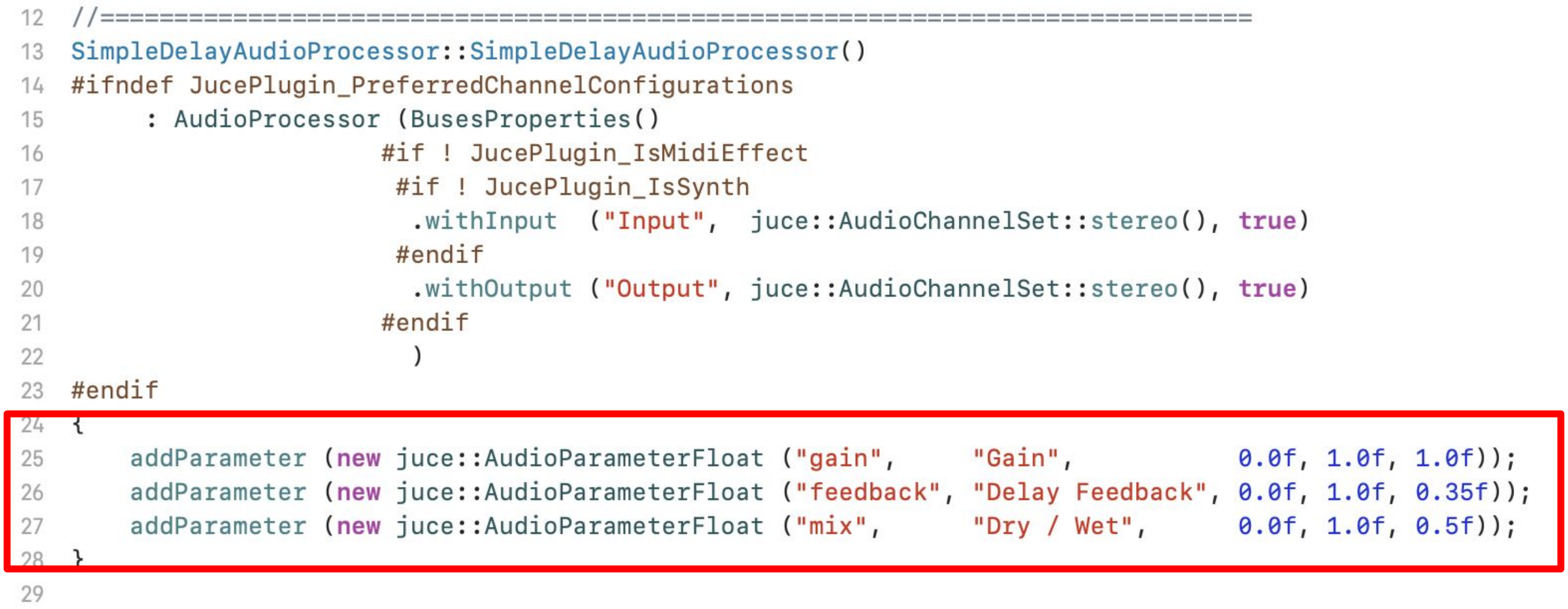
The audio at time *t* is combined with a recursively attenuated audio at time (*t* - *nD*) where *D* is the delay time and *n* = 1,2,3,4,...

Increasing the delay time *D* increases the time between echos

Increasing the attenuation decreases the time taken for the echos to fade away

Our SimpleDelay plug-in will feature a fixed delay time, but a variable feedback (opposite of attenuation) with a dry/wet mix control

#04: Add some more parameters



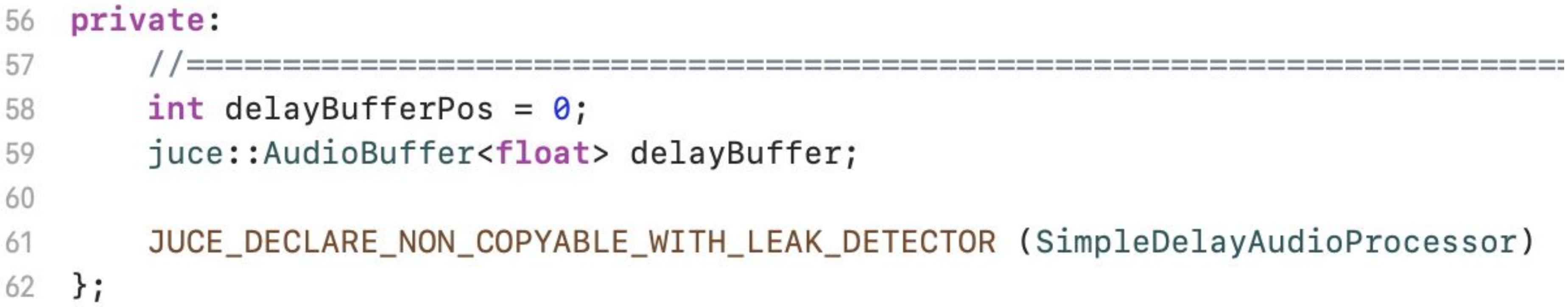
```
//==============================================================================
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
    addParameter (new juce::AudioParameterFloat ("gain",     "Gain",           0.0f, 1.0f, 1.0f));
    addParameter (new juce::AudioParameterFloat ("feedback", "Delay Feedback", 0.0f, 1.0f, 0.35f));
    addParameter (new juce::AudioParameterFloat ("mix",      "Dry / Wet",      0.0f, 1.0f, 0.5f));
}

```

#04: The delay algorithm

A simple circular buffer of audio history

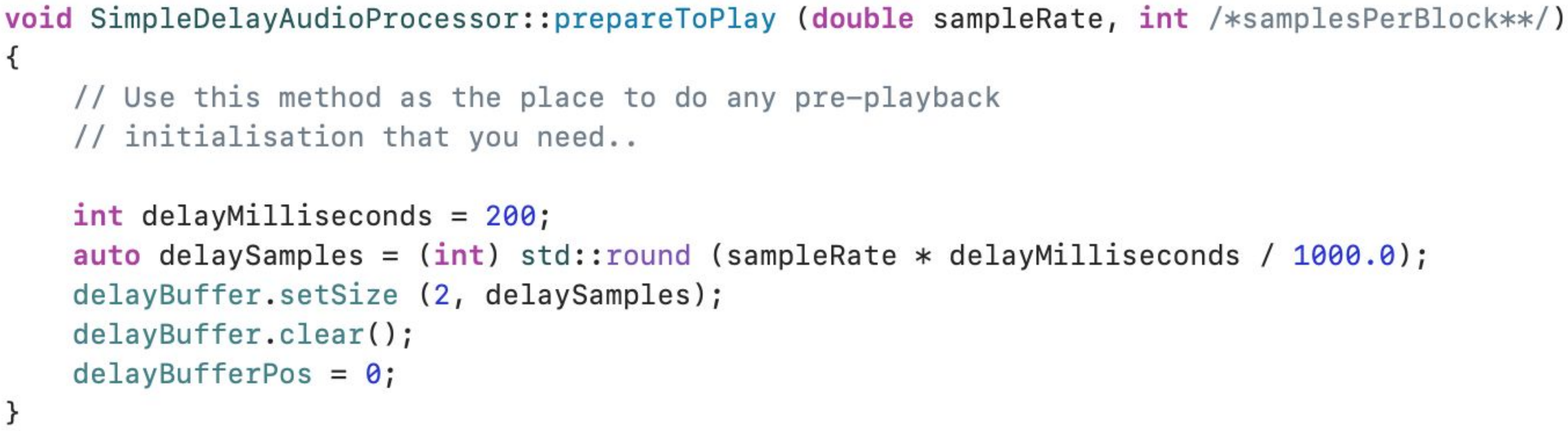
- The size of the circular buffer will determine the (fixed) delay
- Each call to `processBlock` we will add incoming audio to the circular buffer
- The output audio will be a combination of the input and the contents of the circular buffer

#04: Add a circular buffer to `PluginProcessor.h`



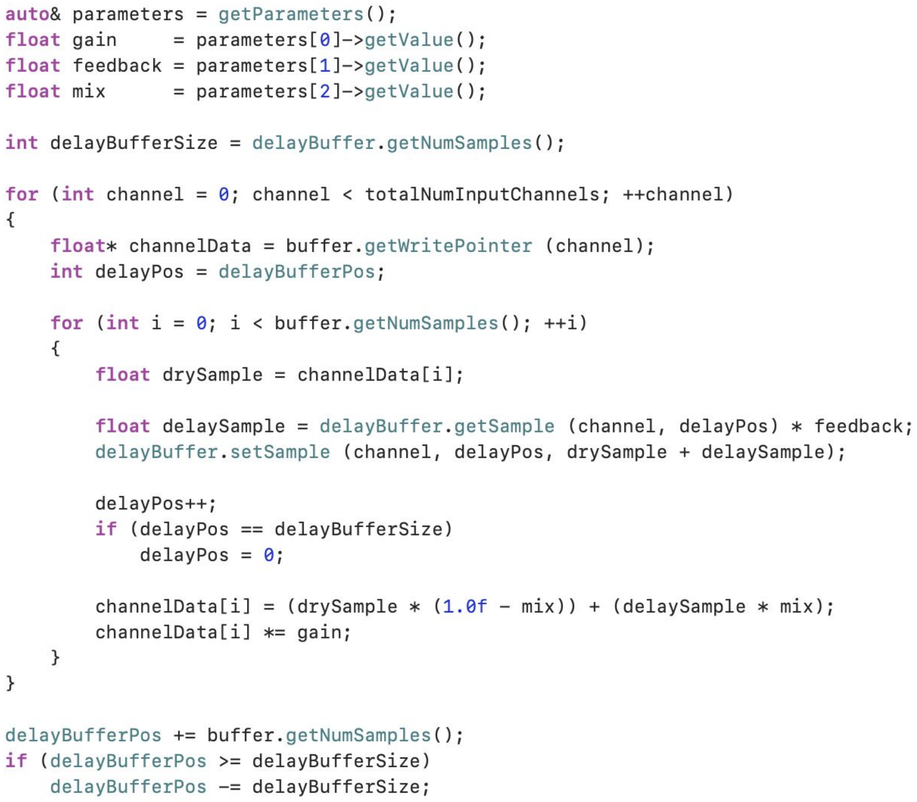
```
private:
    //==================================================================
    int delayBufferPos = 0;
    juce::AudioBuffer<float> delayBuffer;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
};
```

#04: Configure the buffer in `prepareToPlay`



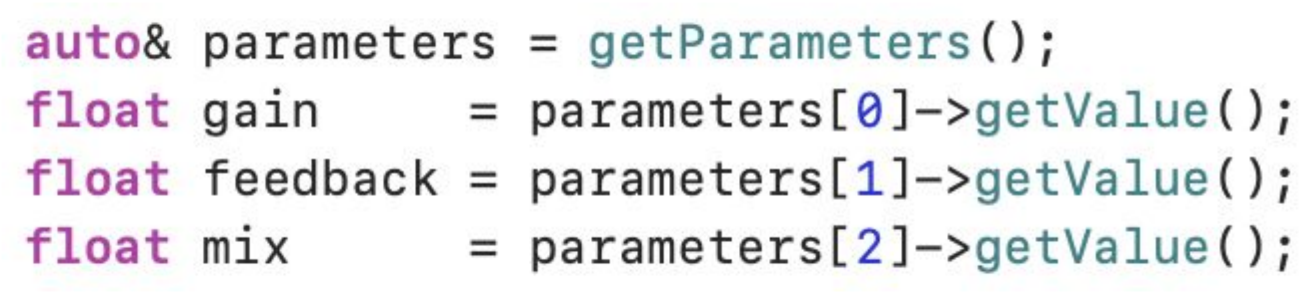
```
void SimpleDelayAudioProcessor::prepareToPlay (double sampleRate, int /*samplesPerBlock**/)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..

    int delayMilliseconds = 200;
    auto delaySamples = (int) std::round (sampleRate * delayMilliseconds / 1000.0);
    delayBuffer.setSize (2, delaySamples);
    delayBuffer.clear();
    delayBufferPos = 0;
}
```

#04: Implement delay algorithm in `processBlock`



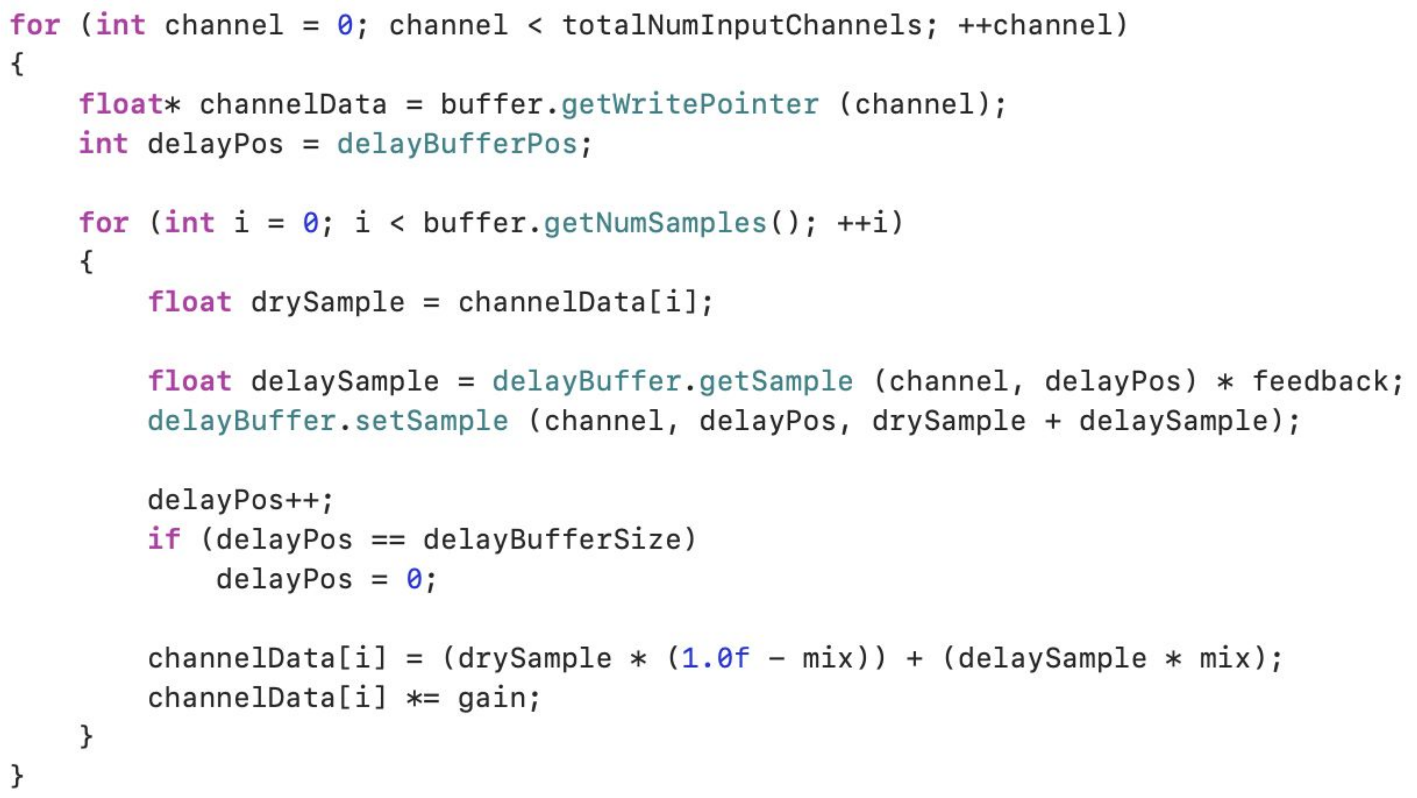
```
auto& parameters = getParameters();
float gain     = parameters[0]->getValue();
float feedback = parameters[1]->getValue();
float mix      = parameters[2]->getValue();

int delayBufferSize = delayBuffer.getNumSamples();

for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    float* channelData = buffer.getWritePointer (channel);
    int delayPos = delayBufferPos;

    for (int i = 0; i < buffer.getNumSamples(); ++i)
    {
        float drySample = channelData[i];

        float delaySample = delayBuffer.getSample (channel, delayPos) * feedback;
        delayBuffer.setSample (channel, delayPos, drySample + delaySample);

        delayPos++;
        if (delayPos == delayBufferSize)
            delayPos = 0;

        channelData[i] = (drySample * (1.0f - mix)) + (delaySample * mix);
        channelData[i] *= gain;
    }
}

delayBufferPos += buffer.getNumSamples();
if (delayBufferPos >= delayBufferSize)
    delayBufferPos -= delayBufferSize;
```

#04: Implement delay algorithm in `processBlock`



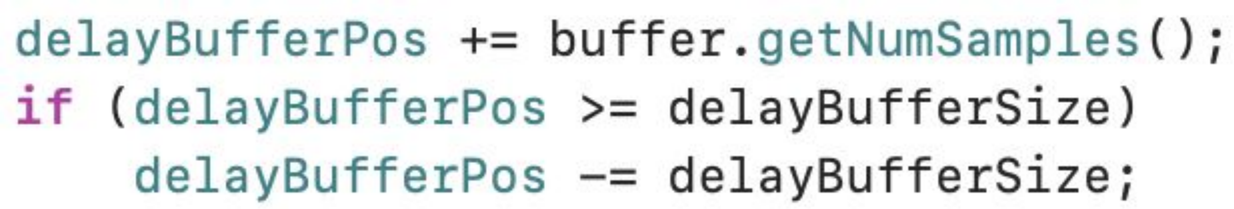
```
auto& parameters = getParameters();
float gain     = parameters[0]->getValue();
float feedback = parameters[1]->getValue();
float mix      = parameters[2]->getValue();
```

#04: Implement a delay algorithm in `processBlock`



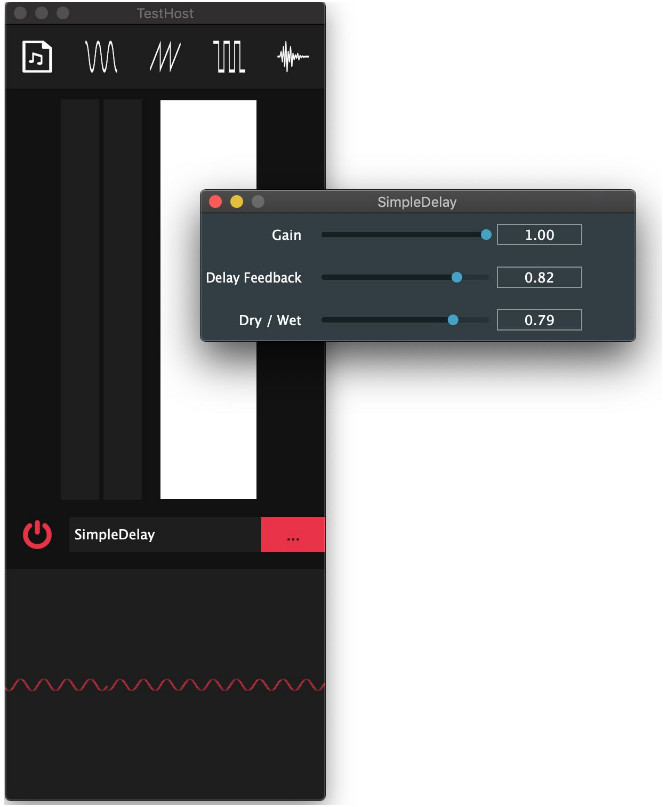
```cpp
for (int channel = 0; channel < totalNumInputChannels; ++channel)
{
    float* channelData = buffer.getWritePointer (channel);
    int delayPos = delayBufferPos;

    for (int i = 0; i < buffer.getNumSamples(); ++i)
    {
        float drySample = channelData[i];

        float delaySample = delayBuffer.getSample (channel, delayPos) * feedback;
        delayBuffer.setSample (channel, delayPos, drySample + delaySample);

        delayPos++;
        if (delayPos == delayBufferSize)
            delayPos = 0;

        channelData[i] = (drySample * (1.0f - mix)) + (delaySample * mix);
        channelData[i] *= gain;
    }
}
```

#04: Implement delay algorithm in `processBlock`



```
delayBufferPos += buffer.getNumSamples();
if (delayBufferPos >= delayBufferSize)
    delayBufferPos -= delayBufferSize;
```

#04: Have a play with the plug-in



SimpleDelay

Gain 1.00

Delay Feedback 0.82

Dry / Wet 0.79

#05: Parameter management and state

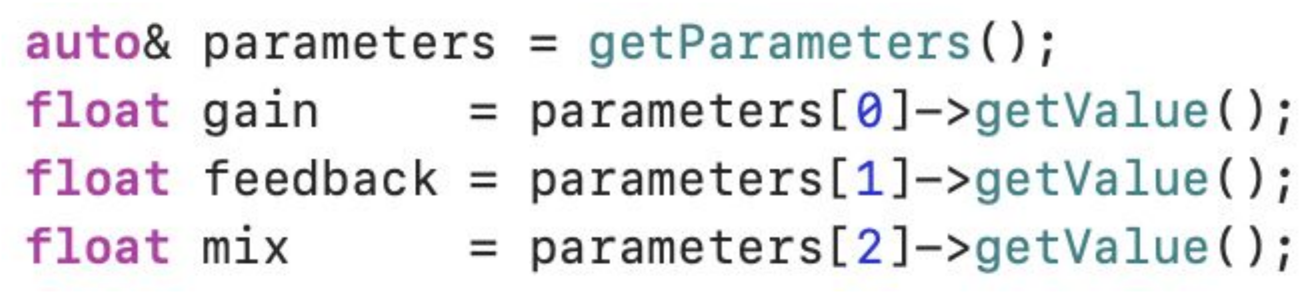
Objectives of this section:

Use an `AudioProcessorValueTreeState` to manage your parameters

- See improved parameter handling
- Serialise and deserialise your plug-ins state

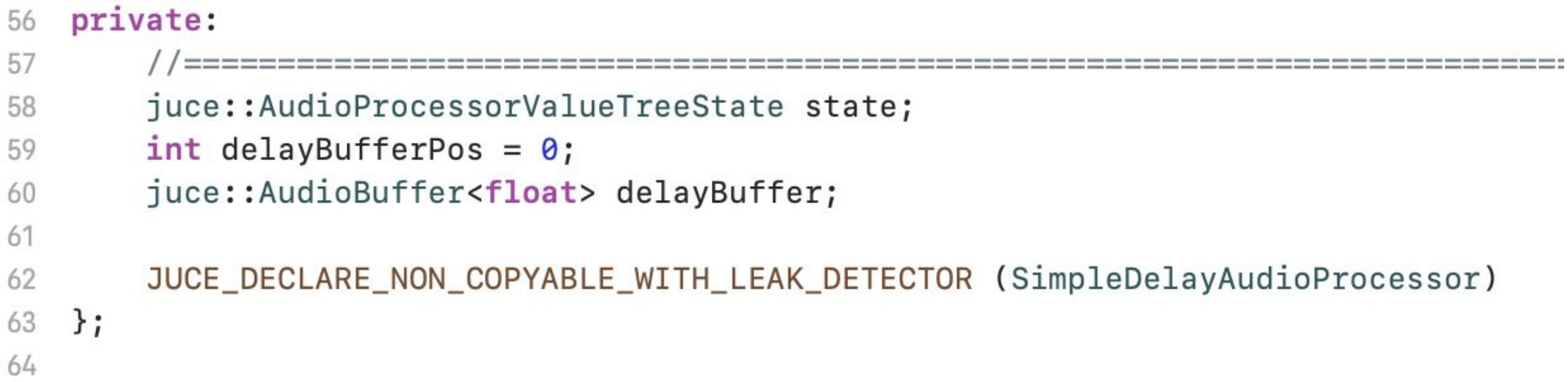
This section requires quite a lot of typing to complete. Start from the contents of the `06` folder and review the changes rather than typing along.

#05: Parameter management



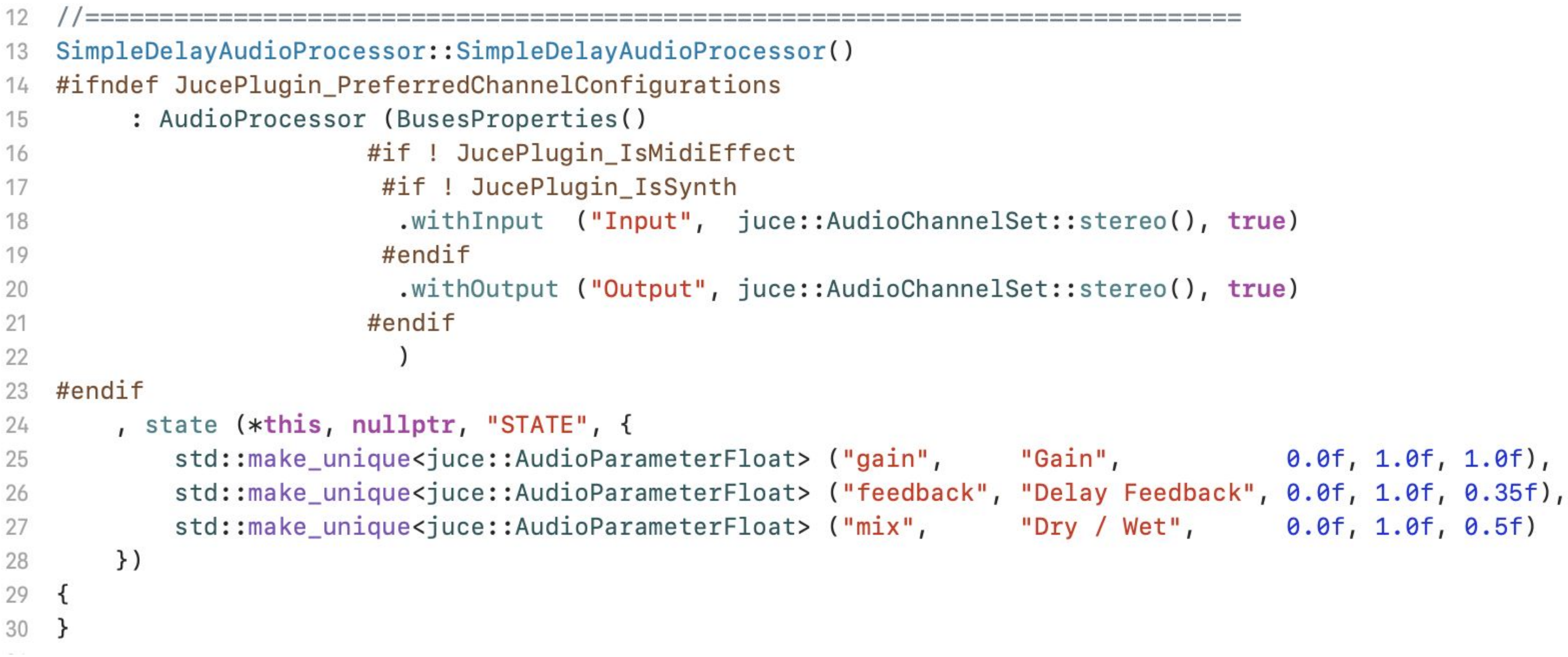
```
auto& parameters = getParameters();
float gain     = parameters[0]->getValue();
float feedback = parameters[1]->getValue();
float mix      = parameters[2]->getValue();
```

#05: Adding an AudioProcessorValueTreeState



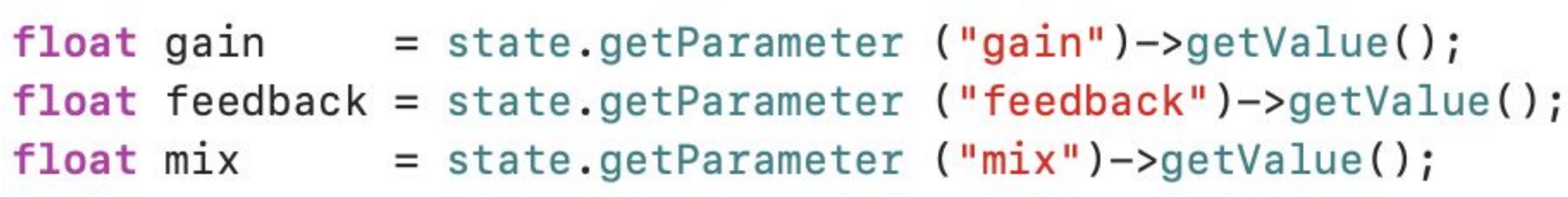
```
private:
    //==========================================================================
    juce::AudioProcessorValueTreeState state;
    int delayBufferPos = 0;
    juce::AudioBuffer<float> delayBuffer;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
};
```

#05: Adding an AudioProcessorValueTreeState



```
//==============================================================================
SimpleDelayAudioProcessor::SimpleDelayAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
    , state (*this, nullptr, "STATE", {
        std::make_unique<juce::AudioParameterFloat> ("gain",     "Gain",           0.0f, 1.0f, 1.0f),
        std::make_unique<juce::AudioParameterFloat> ("feedback", "Delay Feedback", 0.0f, 1.0f, 0.35f),
        std::make_unique<juce::AudioParameterFloat> ("mix",      "Dry / Wet",      0.0f, 1.0f, 0.5f)
    })
{
}
```

#05: Parameter management



```
float gain     = state.getParameter ("gain")->getValue();
float feedback = state.getParameter ("feedback")->getValue();
float mix      = state.getParameter ("mix")->getValue();
```

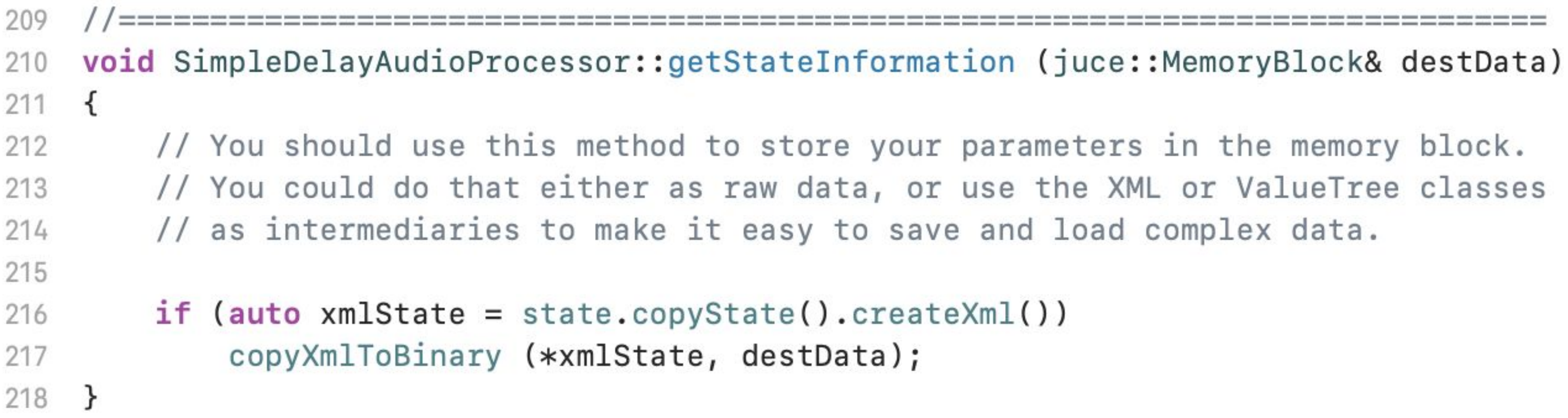
#05: Saving and restoring plug-in state

When plug-in hosts save and load projects, each plug-in must save and restore its state

- `getStateInformation (juce::MemoryBlock& destData)`
- `setStateInformation (const void* data, int sizeInBytes)`

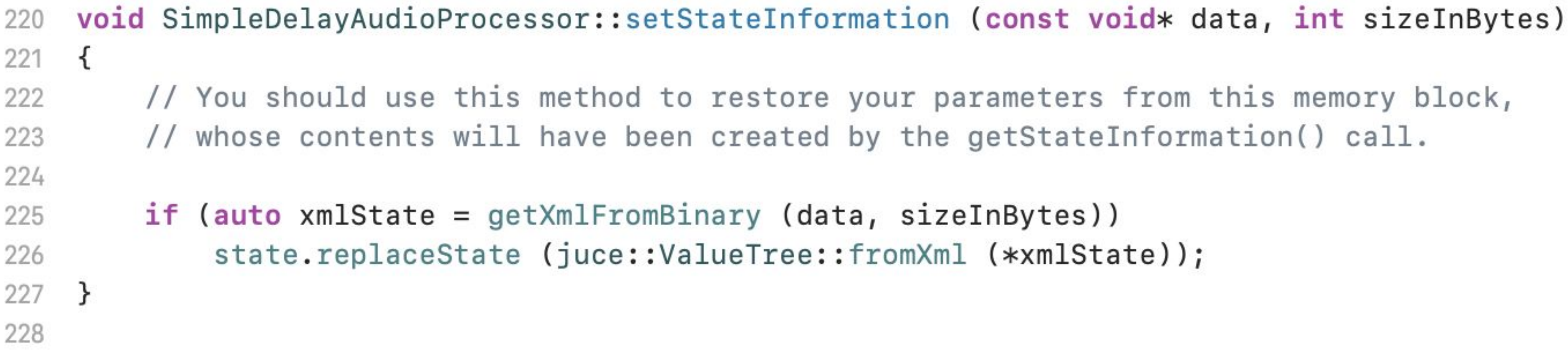
The plug-in's state must be serialised to, and deserialised from, a block of memory managed by the host.

#05: Saving plug-in state



```
//===============================================================================
void SimpleDelayAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    // 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.

    if (auto xmlState = state.copyState().createXml())
        copyXmlToBinary (*xmlState, destData);
}
```

#05: Restoring plug-in state



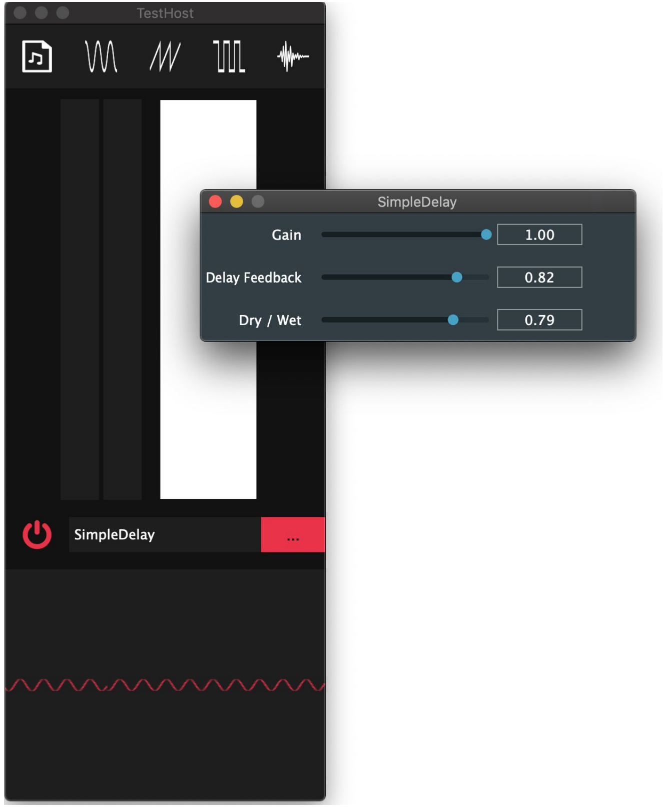
```
void SimpleDelayAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.

    if (auto xmlState = getXmlFromBinary (data, sizeInBytes))
        state.replaceState (juce::ValueTree::fromXml (*xmlState));
}
```

#05: See it working!

- Load the plug-in in the TestHost
- Tweak some parameters
- Unload the plug-in
- Re-load and see the restored state!

Host is calling `getStateInformation` to retrieve and store the state and passes it to the new plug-in instance via `setStateInformation`

#05: Have a play with the plug-in



Gain 1.00

Delay Feedback 0.82

Dry / Wet 0.79

Break

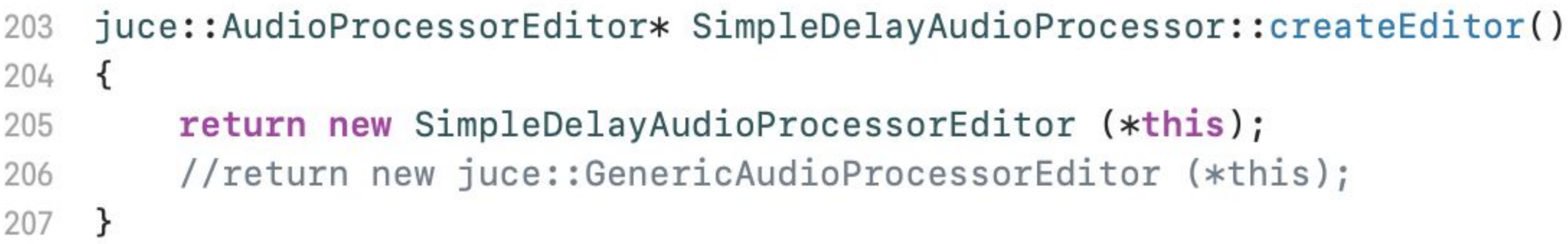
#06: Adding a GUI

Objectives of this section:

Display a custom GUI and draw some graphics

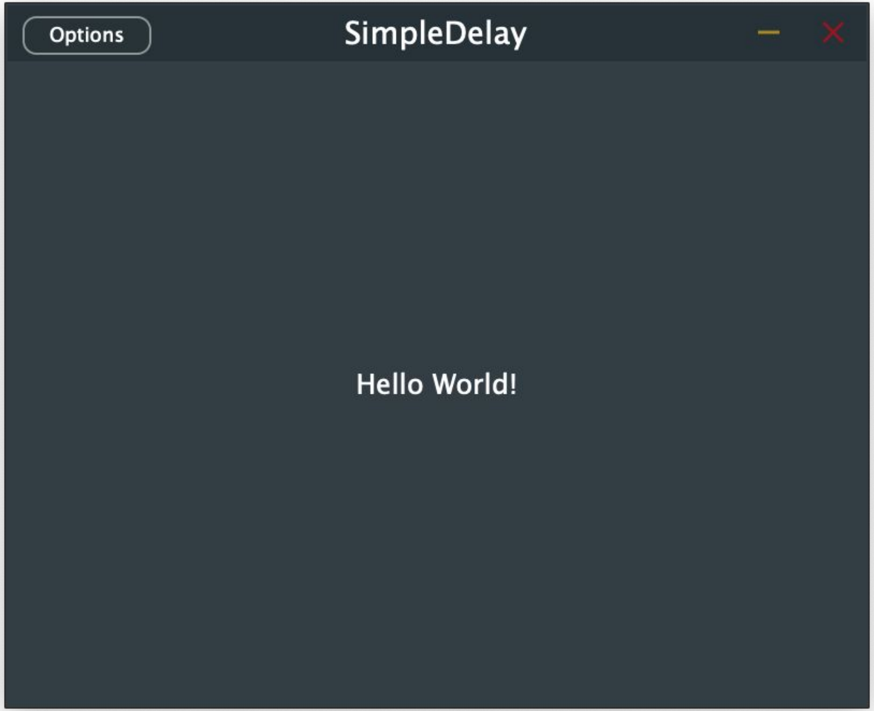
- Basic shapes
- Text

#06: Remove the generic GUI



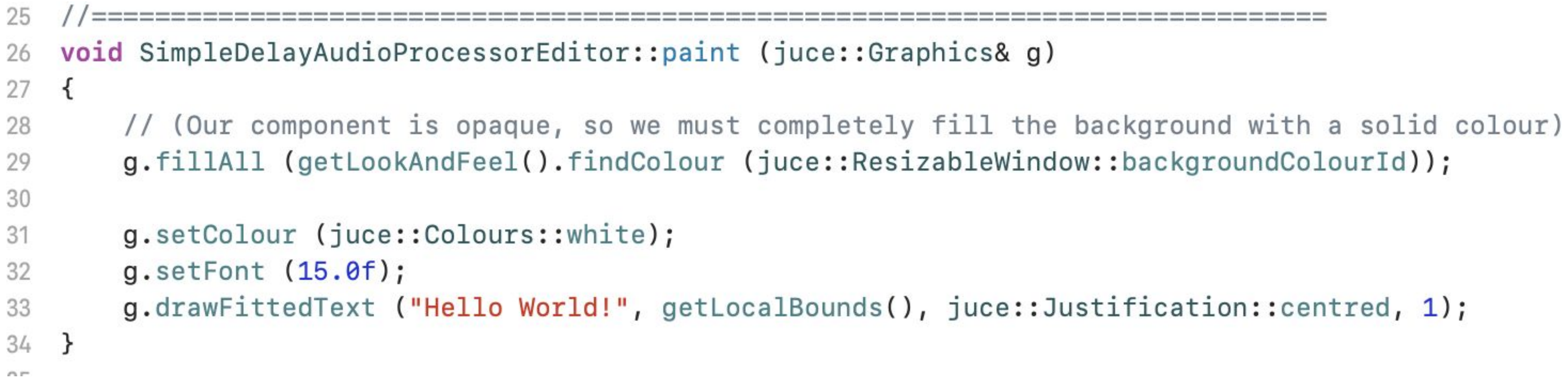
```
juce::AudioProcessorEditor* SimpleDelayAudioProcessor::createEditor()
{
    return new SimpleDelayAudioProcessorEditor (*this);
    //return new juce::GenericAudioProcessorEditor (*this);
}
```

#06: Our custom GUI



Hello World!

#06: Drawing in `paint`



```
//==============================================================================
void SimpleDelayAudioProcessorEditor::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));

    g.setColour (juce::Colours::white);
    g.setFont (15.0f);
    g.drawFittedText ("Hello World!", getLocalBounds(), juce::Justification::centred, 1);
}
```

#06: Threads

Be careful here!

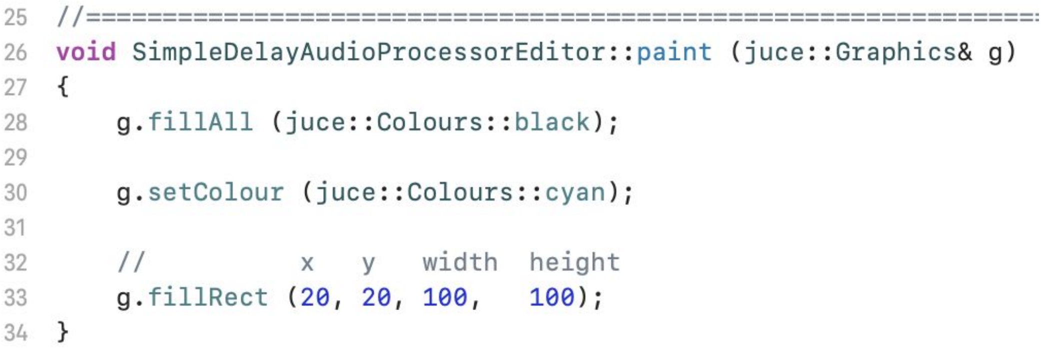
- The GUI is rendered on the "main" (GUI, message) thread
- `processBlock` is called on the audio thread

It's very easy to create race conditions, where one thread is modifying a bit of memory whilst another thread is reading it.

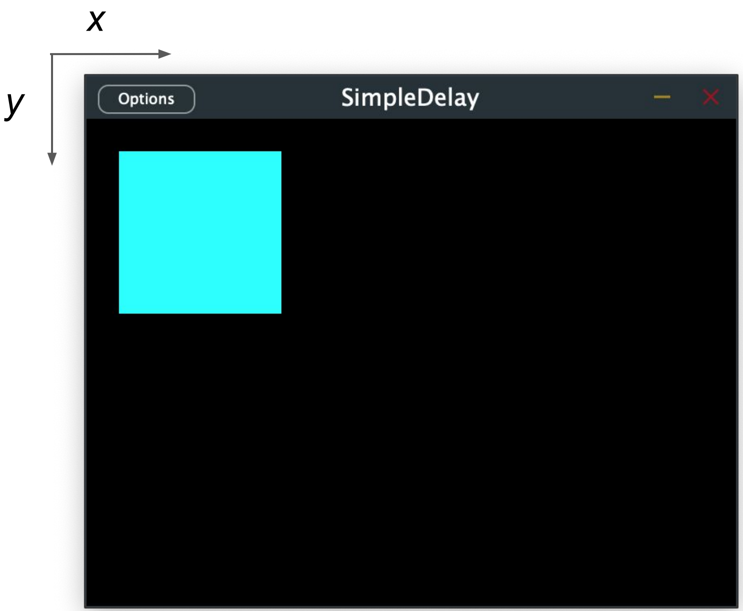
This is easily the most complicated aspect of working with real-time audio.

Using JUCE's `AudioParameter` classes and an `AudioProcessorValueTreeState` makes things much simpler.

#06: Draw a square



```
//==============================================================================
void SimpleDelayAudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll (juce::Colours::black);

    g.setColour (juce::Colours::cyan);

    //          x    y    width   height
    g.fillRect (20, 20, 100,    100);
}
```



#06: Lots of GUI code incoming

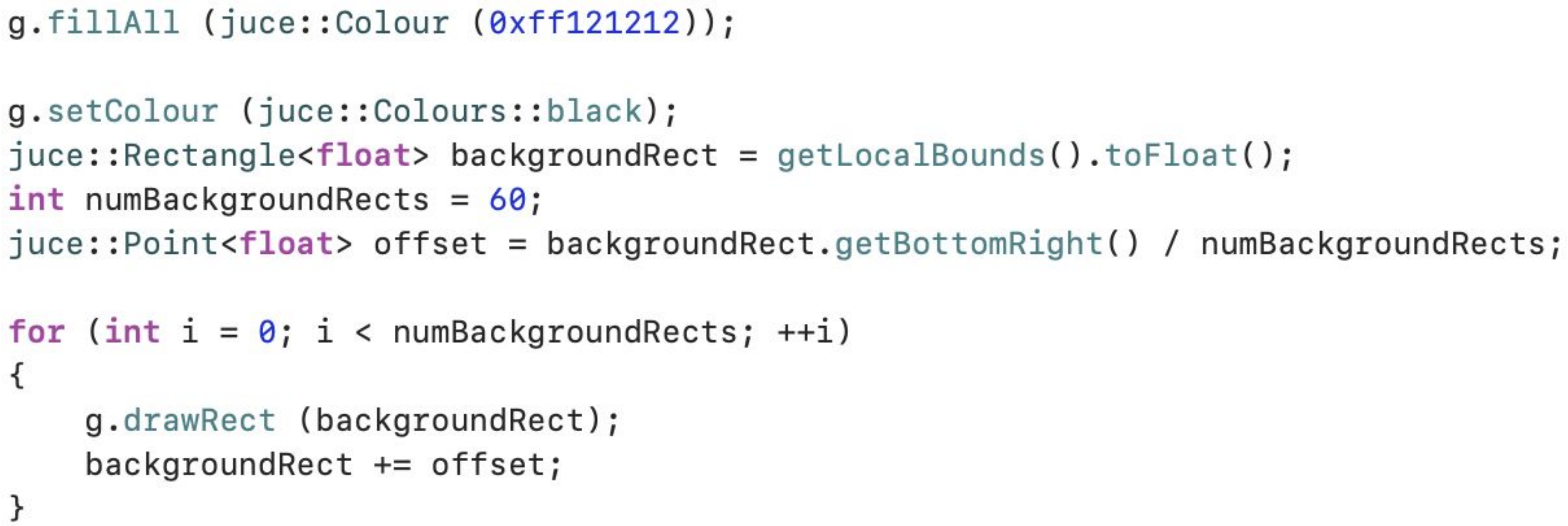
From now until the next break there will be a lot of code to add as we put together a GUI that’s more than a few basic shapes

Don’t worry about keeping up!

During and after the break there will be time to experiment with your own GUIs

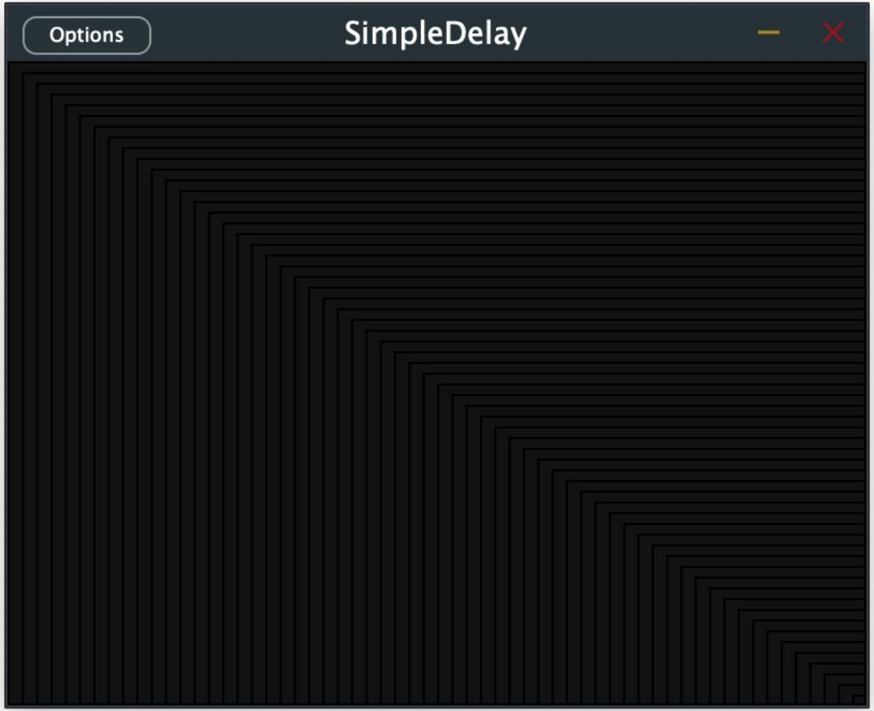
Audio
Developer
Conference

#06: Something more advanced

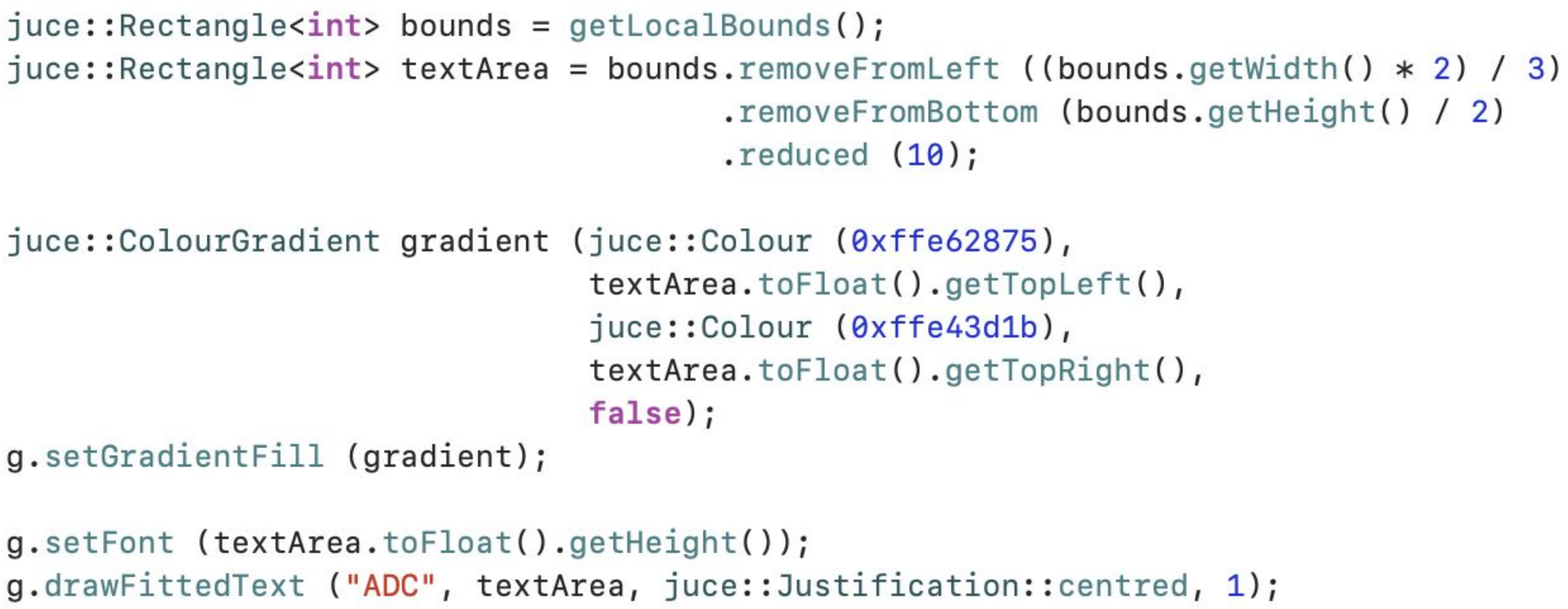


```
g.fillAll (juce::Colour (0xff121212));

g.setColour (juce::Colours::black);
juce::Rectangle<float> backgroundRect = getLocalBounds().toFloat();
int numBackgroundRects = 60;
juce::Point<float> offset = backgroundRect.getBottomRight() / numBackgroundRects;

for (int i = 0; i < numBackgroundRects; ++i)
{
    g.drawRect (backgroundRect);
    backgroundRect += offset;
}
```

#06: Something more advanced

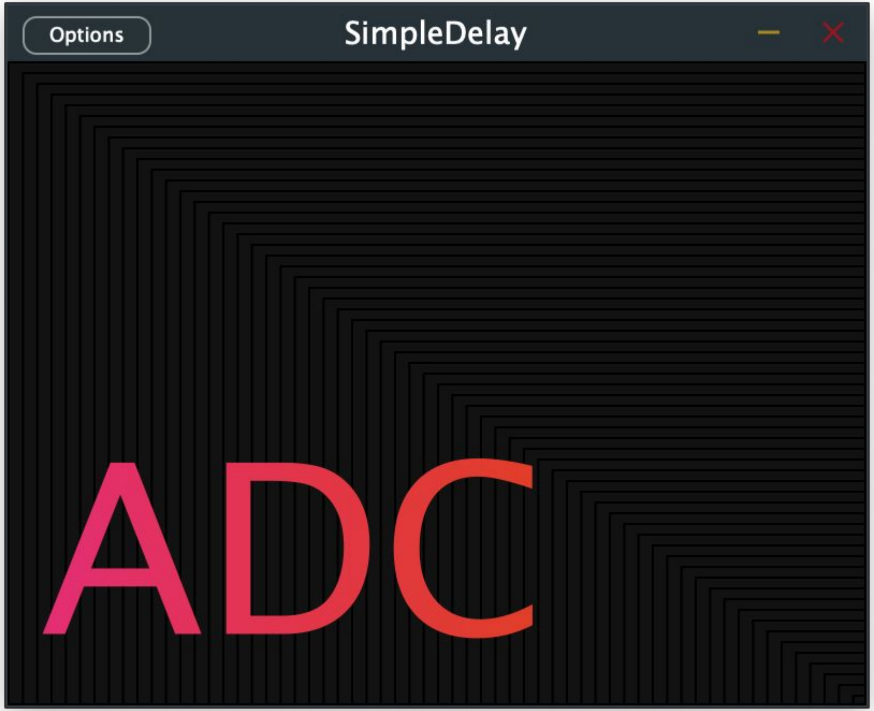


#06: Something more advanced



```
juce::Rectangle<int> bounds = getLocalBounds();
juce::Rectangle<int> textArea = bounds.removeFromLeft ((bounds.getWidth() * 2) / 3)
                                      .removeFromBottom (bounds.getHeight() / 2)
                                      .reduced (10);

juce::ColourGradient gradient (juce::Colour (0xffe62875),
                               textArea.toFloat().getTopLeft(),
                               juce::Colour (0xffe43d1b),
                               textArea.toFloat().getTopRight(),
                               false);
g.setGradientFill (gradient);

g.setFont (textArea.toFloat().getHeight());
g.drawFittedText ("ADC", textArea, juce::Justification::centred, 1);
```

#06: Something more advanced



#07: Components

Objectives of this section:

Create some interactive GUI elements

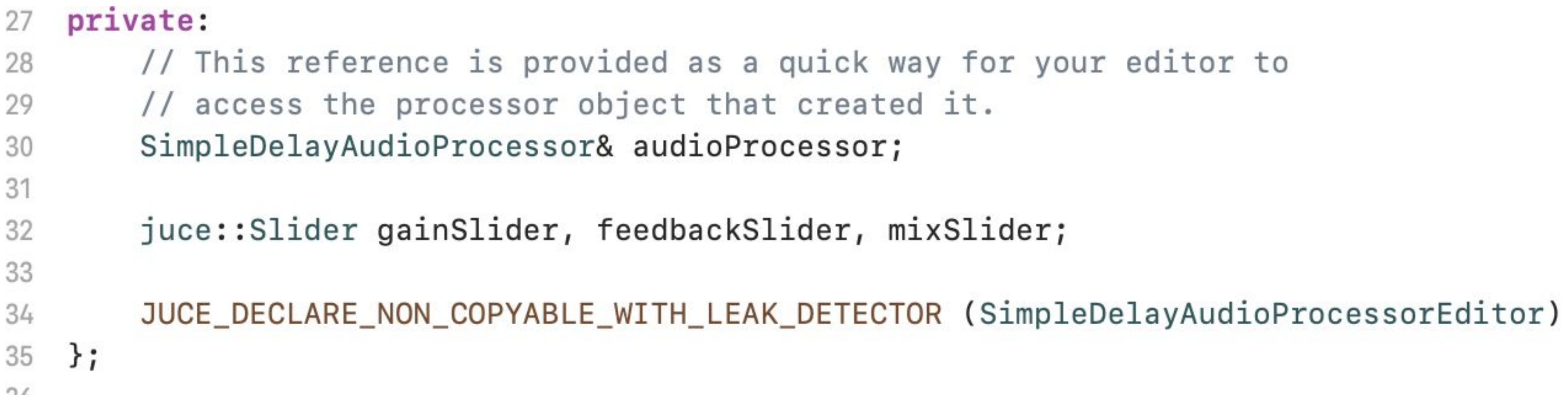
- Add some sliders
- Handle layout changes in `resized`

#07: JUCE Components

JUCE GUIs are trees of components

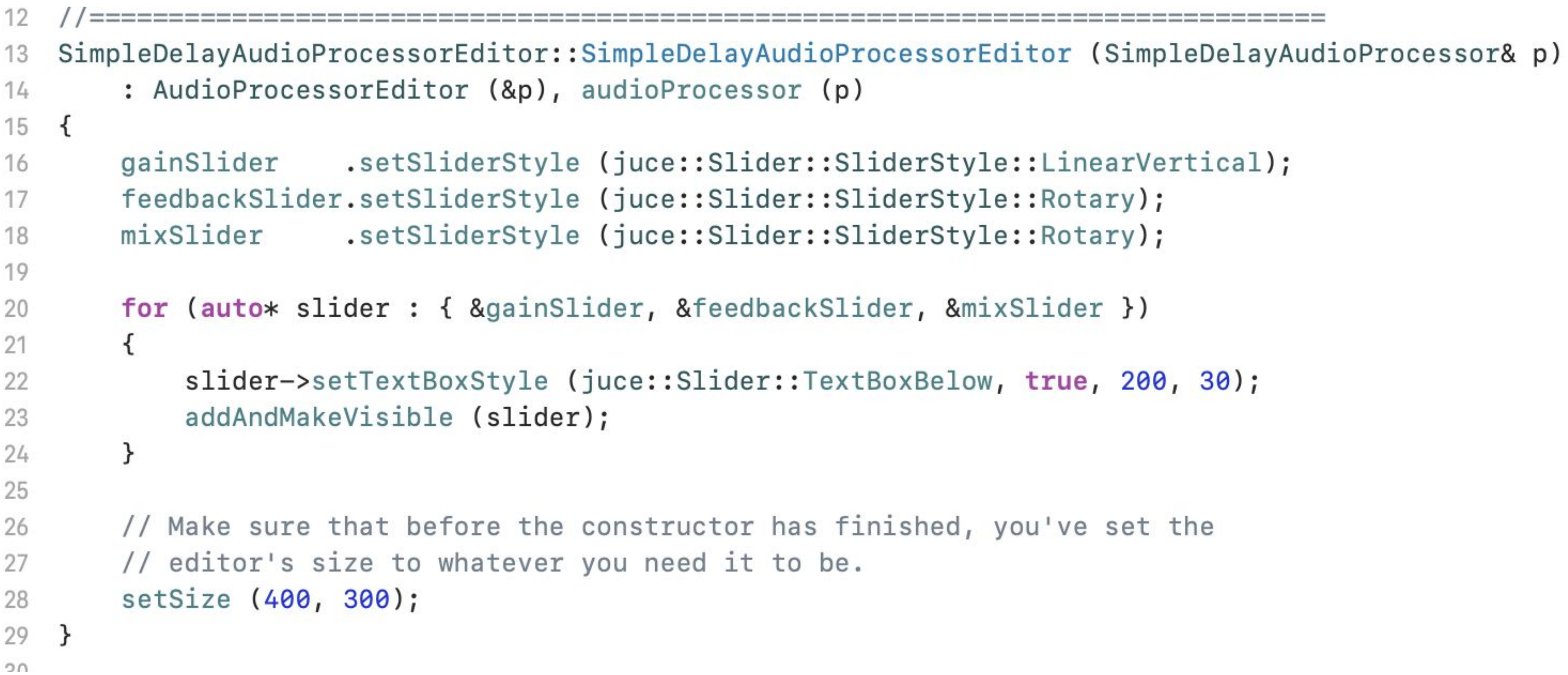
- You can create your own; the `AudioProcessorEditor` is a `Component`
- Parent components are responsible for laying out child components
- Mouse events and keyboard focus can be passed between them
- JUCE has a selection of common widget components you can use

#07: Add some sliders to `PluginEditor.h`



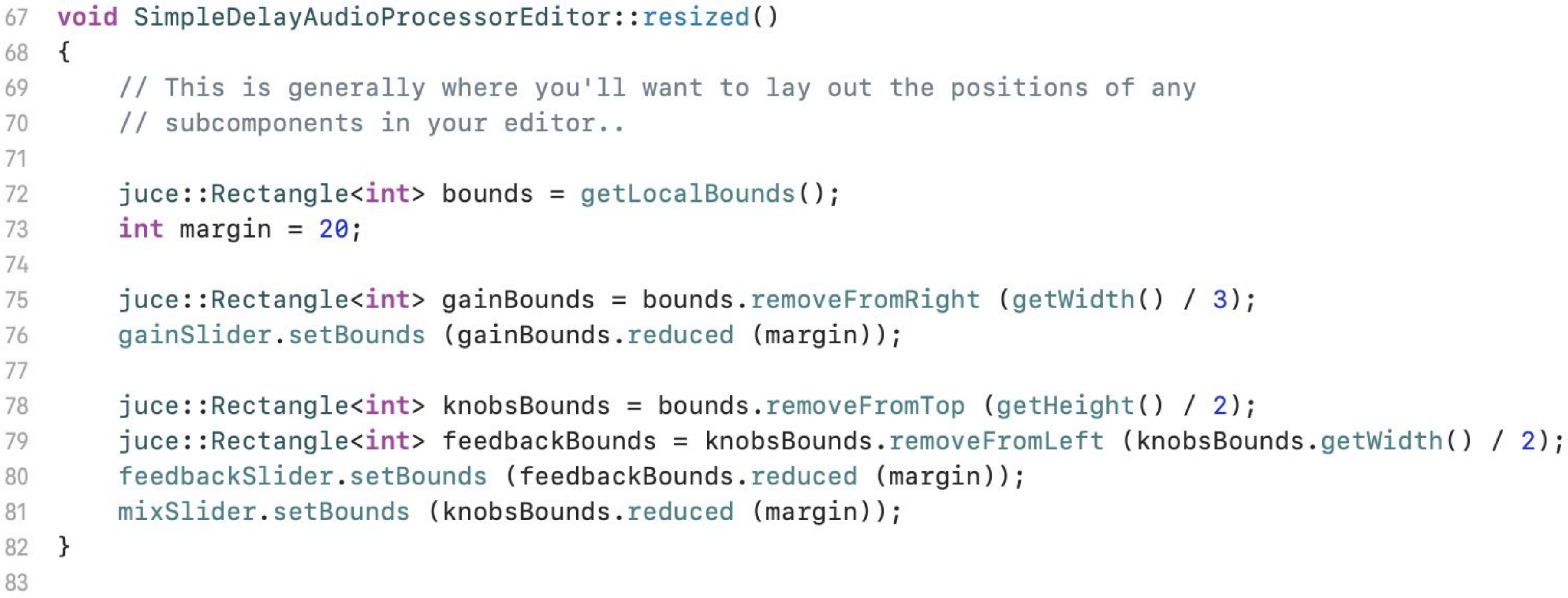
```
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    SimpleDelayAudioProcessor& audioProcessor;

    juce::Slider gainSlider, feedbackSlider, mixSlider;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessorEditor)
};
```

#07: Configure the sliders in `PluginEditor.cpp`



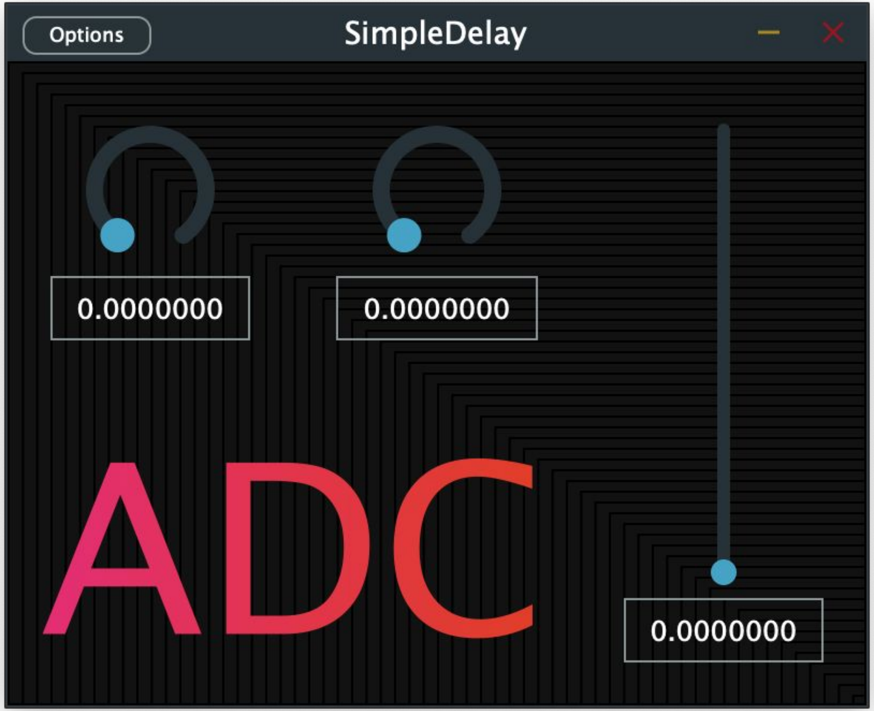
```cpp
//==============================================================================
SimpleDelayAudioProcessorEditor::SimpleDelayAudioProcessorEditor (SimpleDelayAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    gainSlider    .setSliderStyle (juce::Slider::SliderStyle::LinearVertical);
    feedbackSlider.setSliderStyle (juce::Slider::SliderStyle::Rotary);
    mixSlider     .setSliderStyle (juce::Slider::SliderStyle::Rotary);

    for (auto* slider : { &gainSlider, &feedbackSlider, &mixSlider })
    {
        slider->setTextBoxStyle (juce::Slider::TextBoxBelow, true, 200, 30);
        addAndMakeVisible (slider);
    }

    // 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);
}
```

#07: Layout the sliders in `resized`



```cpp
void SimpleDelayAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
    // subcomponents in your editor..

    juce::Rectangle<int> bounds = getLocalBounds();
    int margin = 20;

    juce::Rectangle<int> gainBounds = bounds.removeFromRight (getWidth() / 3);
    gainSlider.setBounds (gainBounds.reduced (margin));

    juce::Rectangle<int> knobsBounds = bounds.removeFromTop (getHeight() / 2);
    juce::Rectangle<int> feedbackBounds = knobsBounds.removeFromLeft (knobsBounds.getWidth() / 2);
    feedbackSlider.setBounds (feedbackBounds.reduced (margin));
    mixSlider.setBounds (knobsBounds.reduced (margin));
}

```

#07: GUI with Sliders



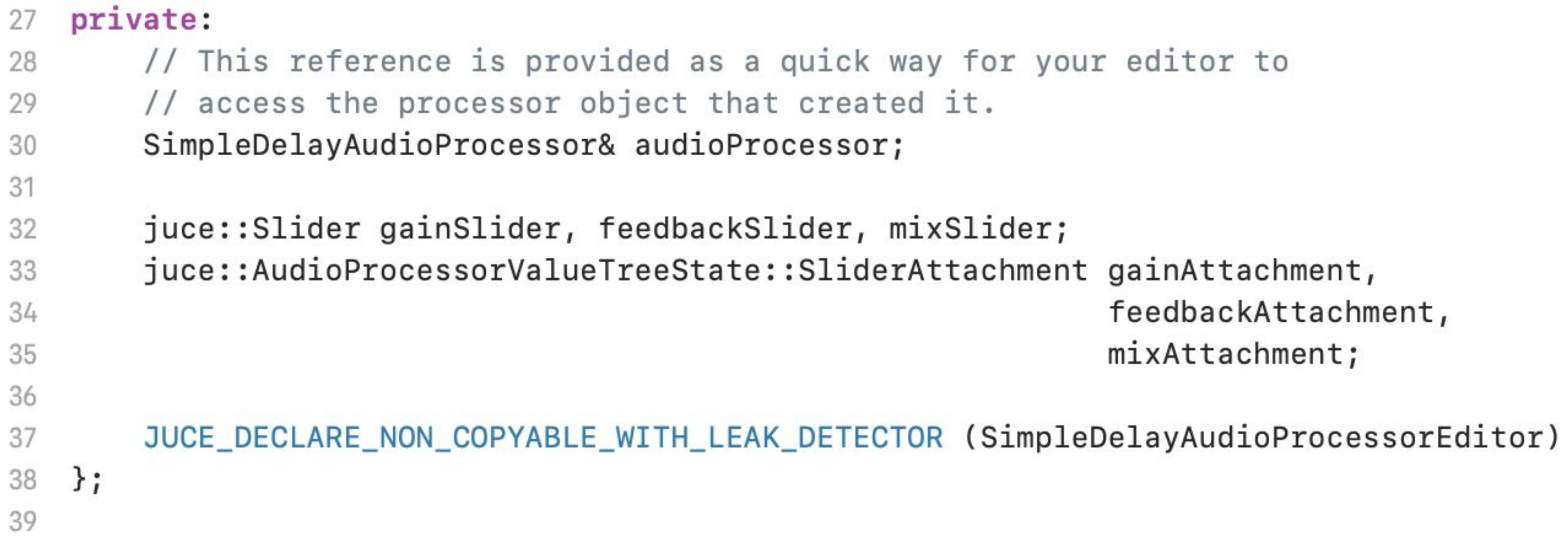
#08: Connecting GUI controls to plug-in parameters

Objectives of this section:

Control the plug-in from the GUI

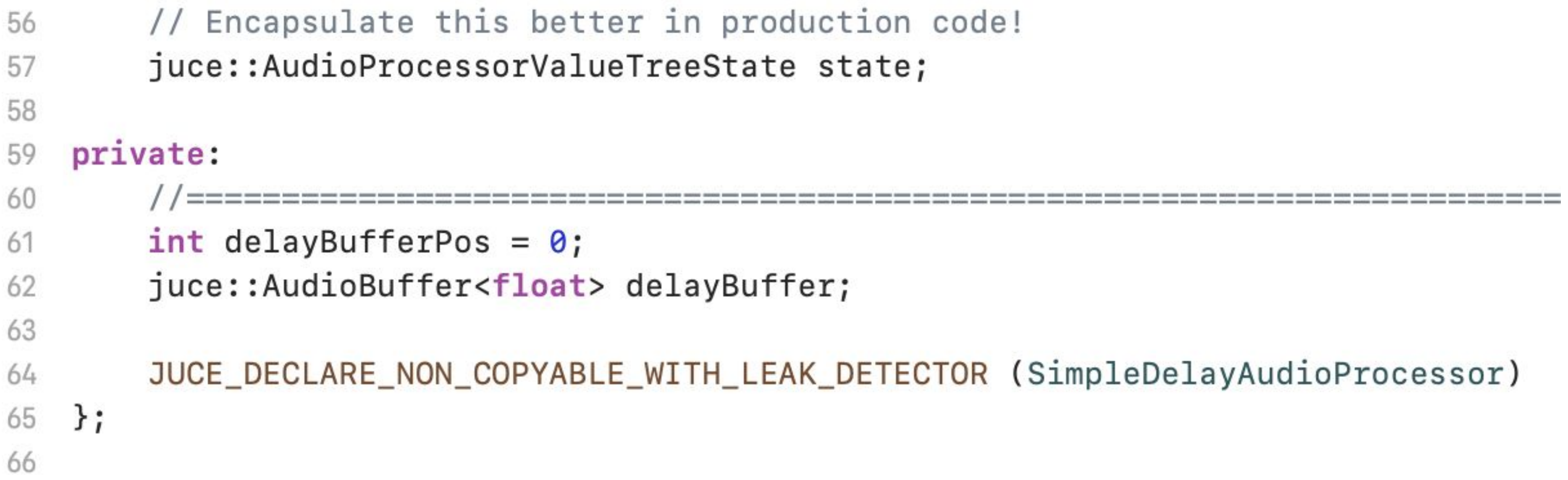
- Use the `AudioProcessorValueTreeState` attachment classes to link `Sliders` to plug-in parameters

#08: Add some attachments to `PluginEditor.h`



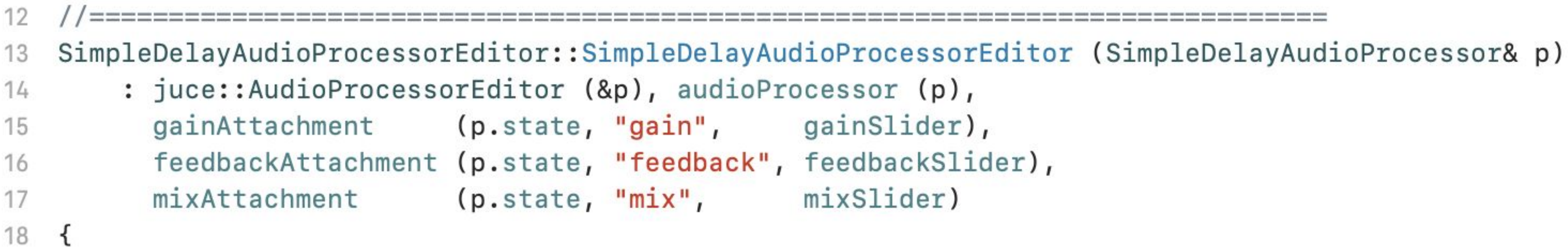
```
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    SimpleDelayAudioProcessor& audioProcessor;

    juce::Slider gainSlider, feedbackSlider, mixSlider;
    juce::AudioProcessorValueTreeState::SliderAttachment gainAttachment,
                                                         feedbackAttachment,
                                                         mixAttachment;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessorEditor)
};

```

#08: We need to pass the state to our editor



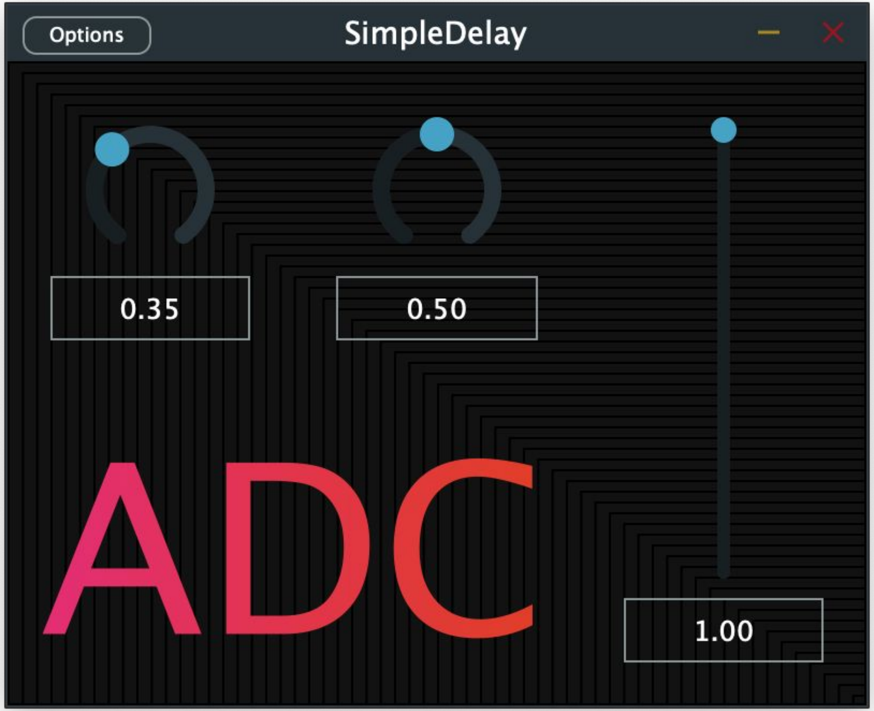
```
    // Encapsulate this better in production code!
    juce::AudioProcessorValueTreeState state;

private:
    //========================================================================
    int delayBufferPos = 0;
    juce::AudioBuffer<float> delayBuffer;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SimpleDelayAudioProcessor)
};

```

#08: Linking `Slider` to plug-in parameters



```
//==============================================================================
SimpleDelayAudioProcessorEditor::SimpleDelayAudioProcessorEditor (SimpleDelayAudioProcessor& p)
    : juce::AudioProcessorEditor (&p), audioProcessor (p),
      gainAttachment     (p.state, "gain",     gainSlider),
      feedbackAttachment (p.state, "feedback", feedbackSlider),
      mixAttachment      (p.state, "mix",      mixSlider)
{
```

#09: The interactive plug-in



Break

TESTING IN HOSTS

Debugging

What is a debugger?

- The most useful tool in a programmer’s arsenal!
- GDB, LLDB, Microsoft Visual Studio Debugger
- CLI/GUI
- Examine program state, pause when conditions are met

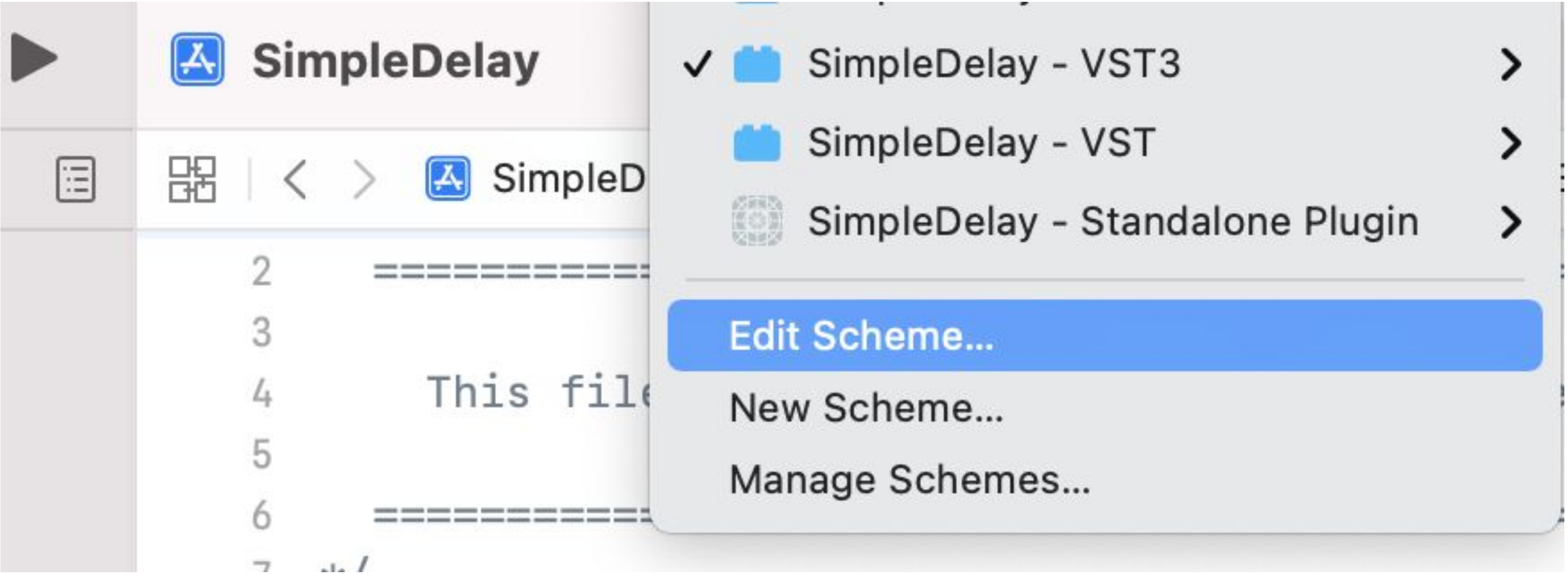
What is a breakpoint?

- Can be set via the CLI or GUI
- Program execution pauses when hit

Debugging

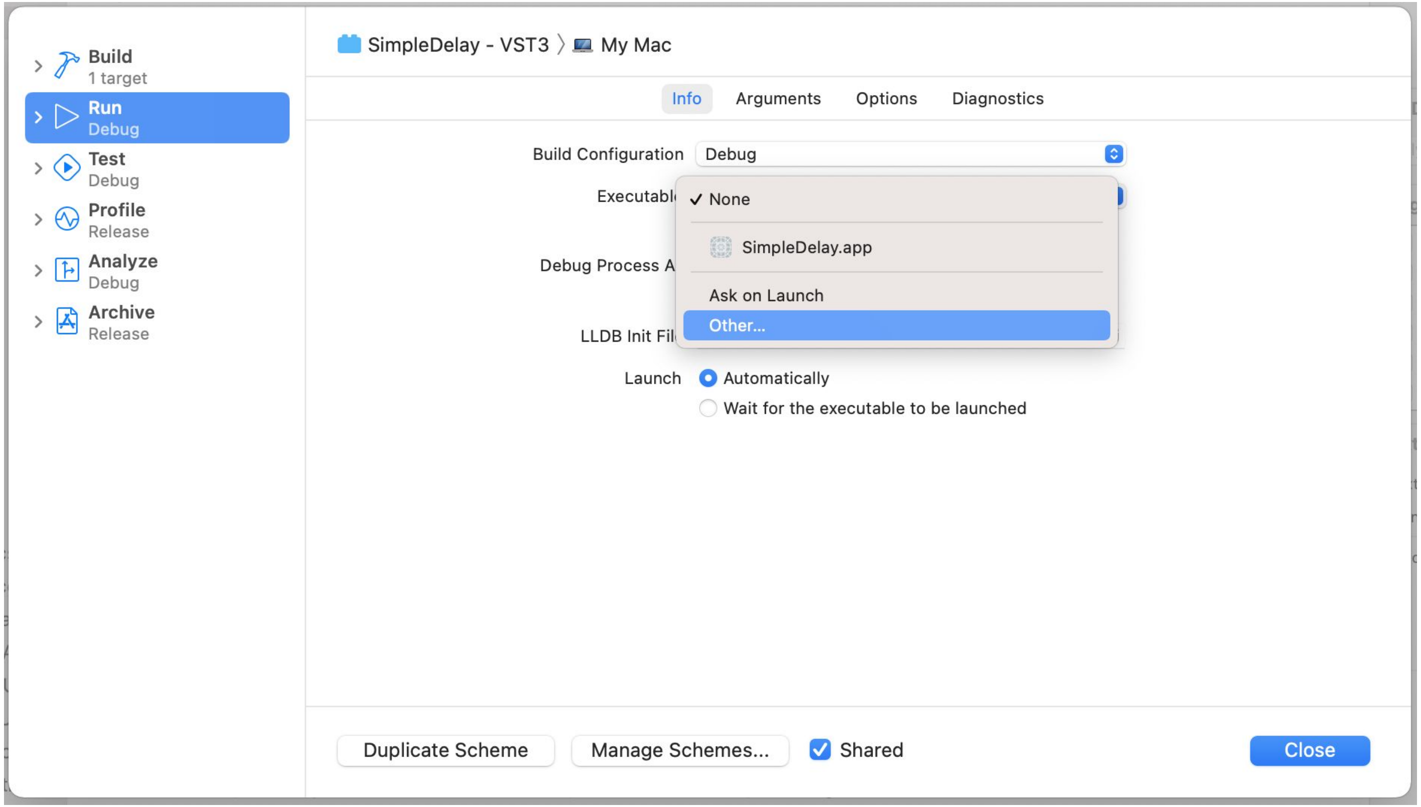
- Debugging standalone plug-in is simple as we control the whole process
- Debugging the plug-in inside an actual host is slightly more complicated as we are running inside a different process (the host)
- Debuggers can *attach* to a separate process to allow you to debug and set breakpoints in your code

macOS



SimpleDelay
SimpleD
2
3
4 This file
5
6
✓ SimpleDelay - VST3
SimpleDelay - VST
SimpleDelay - Standalone Plugin
Edit Scheme...
New Scheme...
Manage Schemes...

macOS



Build Configuration: Debug

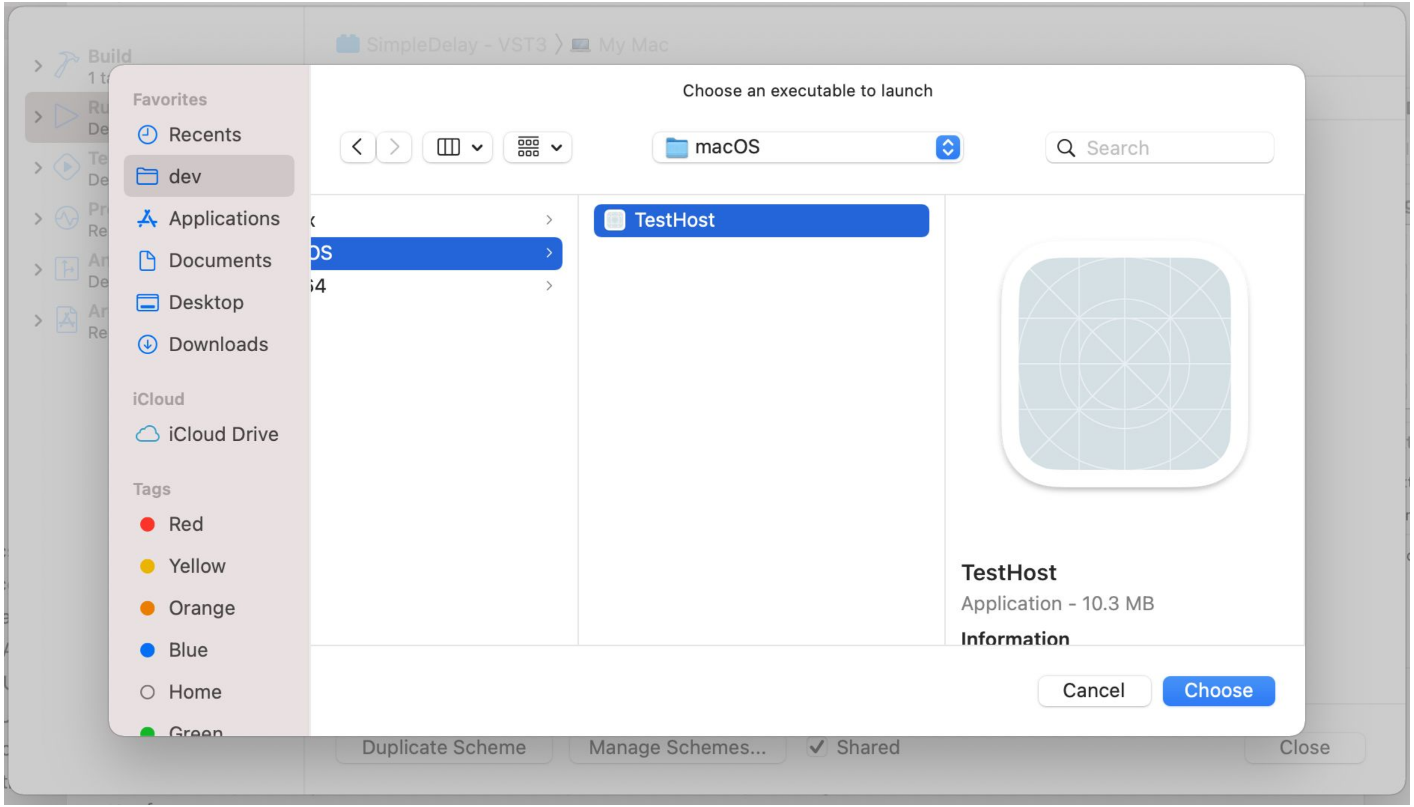
Executabl

Debug Process A

LLDB Init Fil

Launch
● Automatically
○ Wait for the executable to be launched

macOS

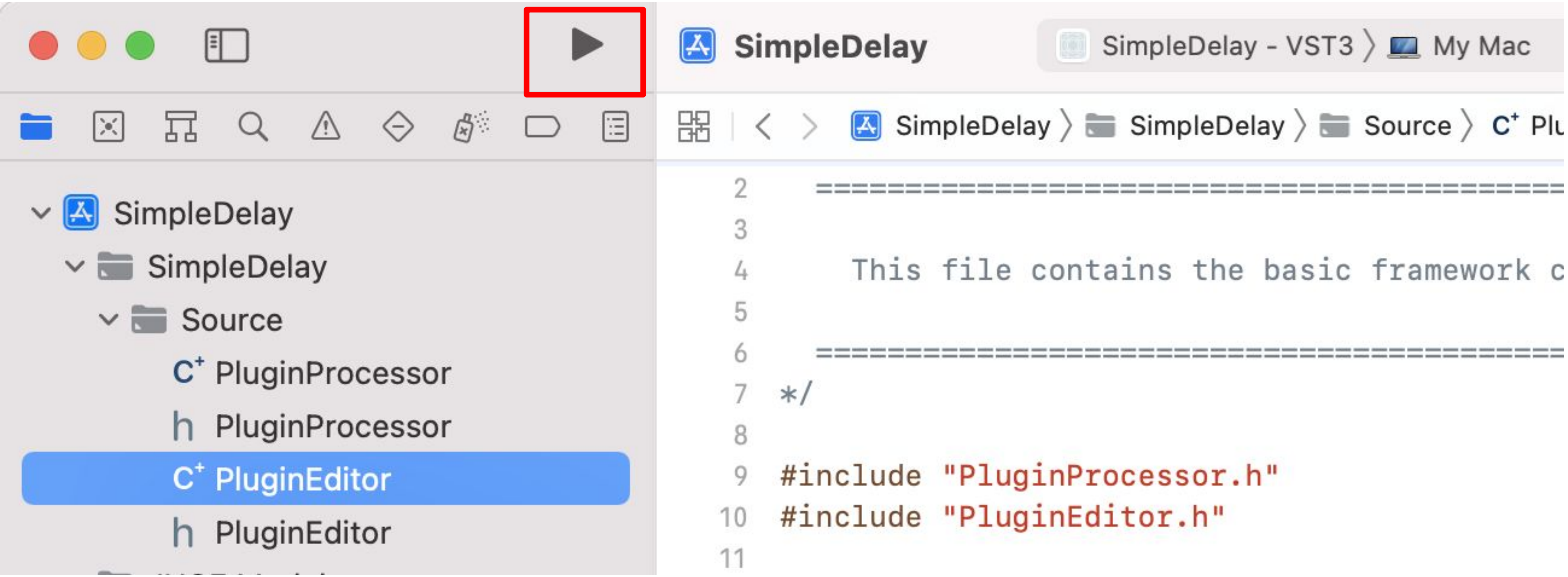


Choose an executable to launch

TestHost

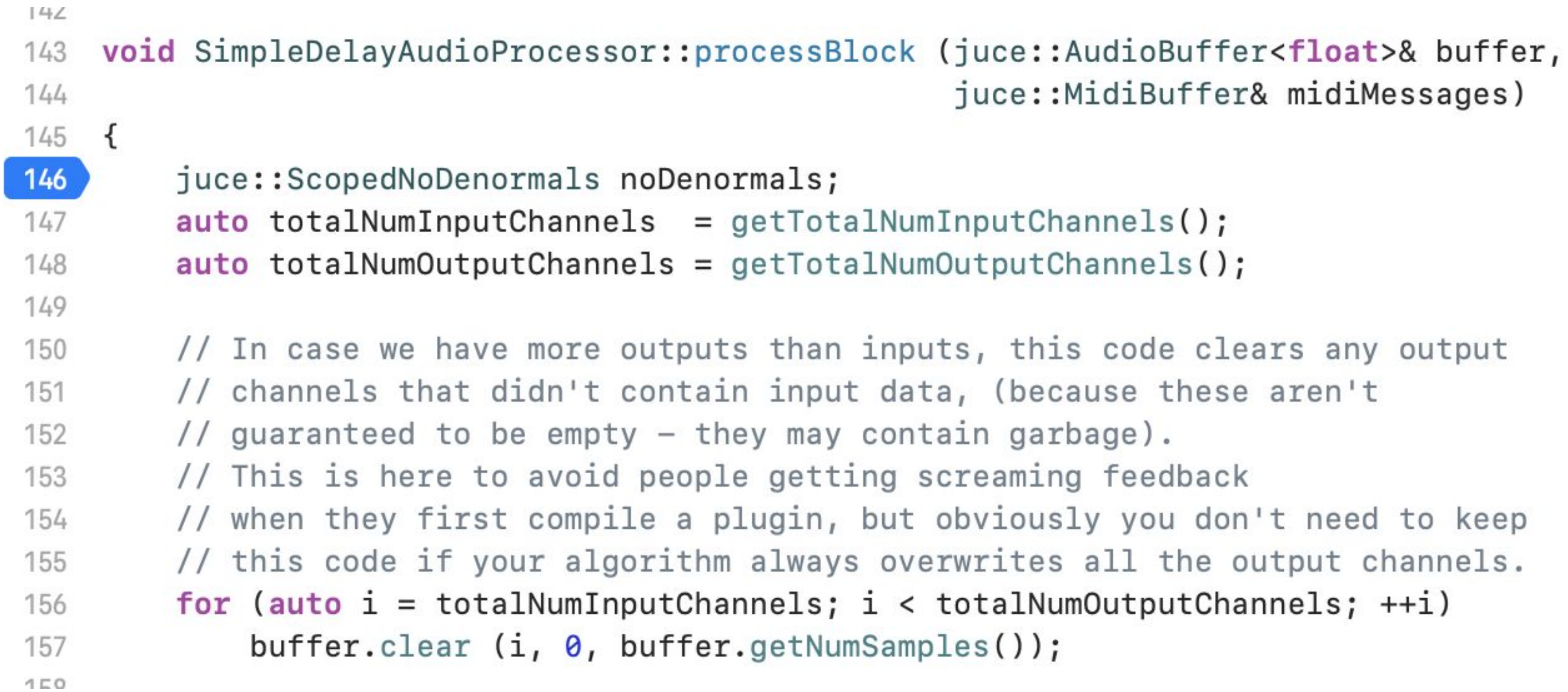
Application - 10.3 MB

macOS



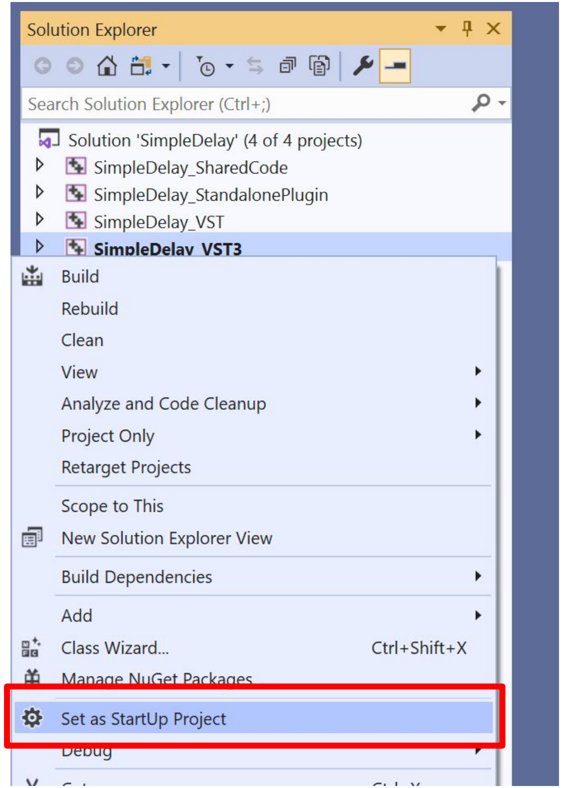
```
  ==========================================
    This file contains the basic framework c

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

#include "PluginProcessor.h"
#include "PluginEditor.h"

```

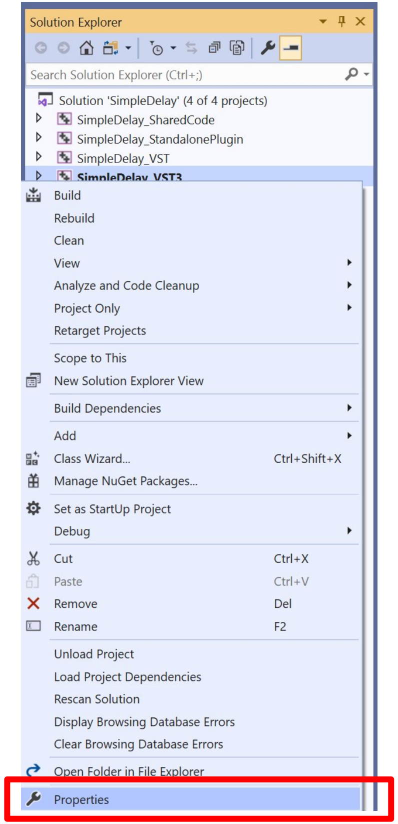
macOS



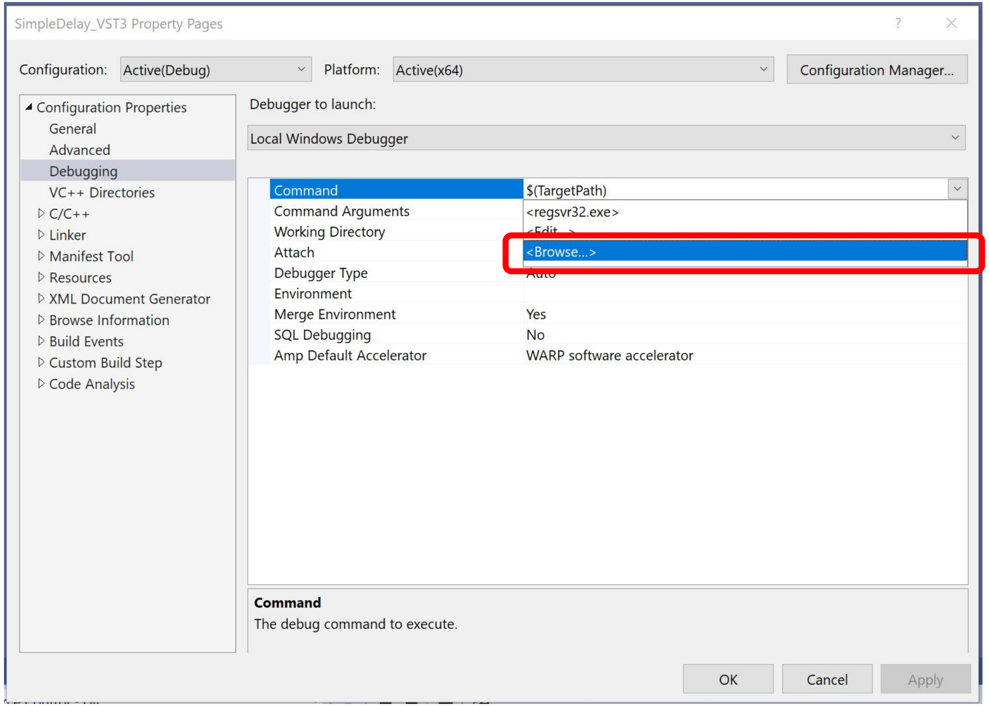
```
void SimpleDelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                              juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = 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());
```

Windows


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'SimpleDelay' (4 of 4 projects)
SimpleDelay_SharedCode
SimpleDelay_StandalonePlugin
SimpleDelay_VST
SimpleDelay_VST3
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Retarget Projects
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages
Set as StartUp Project
Debug


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'SimpleDelay' (4 of 4 projects)
SimpleDelay_SharedCode
SimpleDelay_StandalonePlugin
SimpleDelay_VST
SimpleDelay_VST3
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Retarget Projects
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Set as StartUp Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Load Project Dependencies
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Open Folder in File Explorer
Properties

Windows



Configuration: Active(Debug)

Platform: Active(x64)

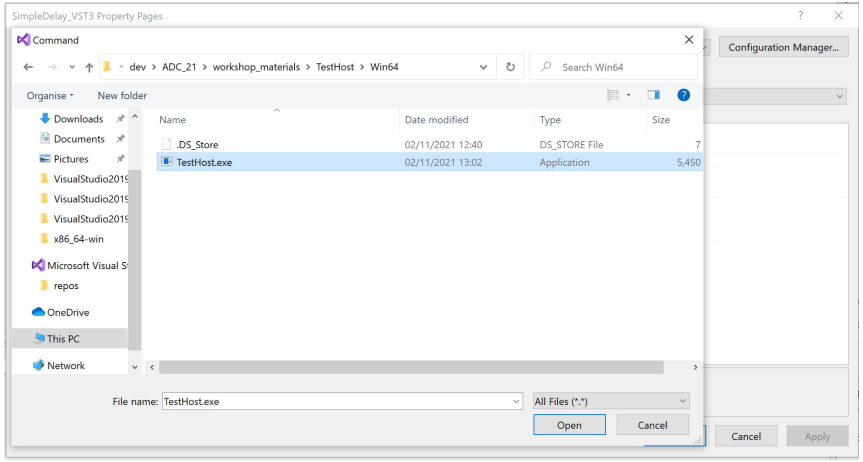
Debugger to launch:

Local Windows Debugger

Command	$(TargetPath)
Command Arguments	<regsvr32.exe>
Working Directory	
Attach	
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command

The debug command to execute.



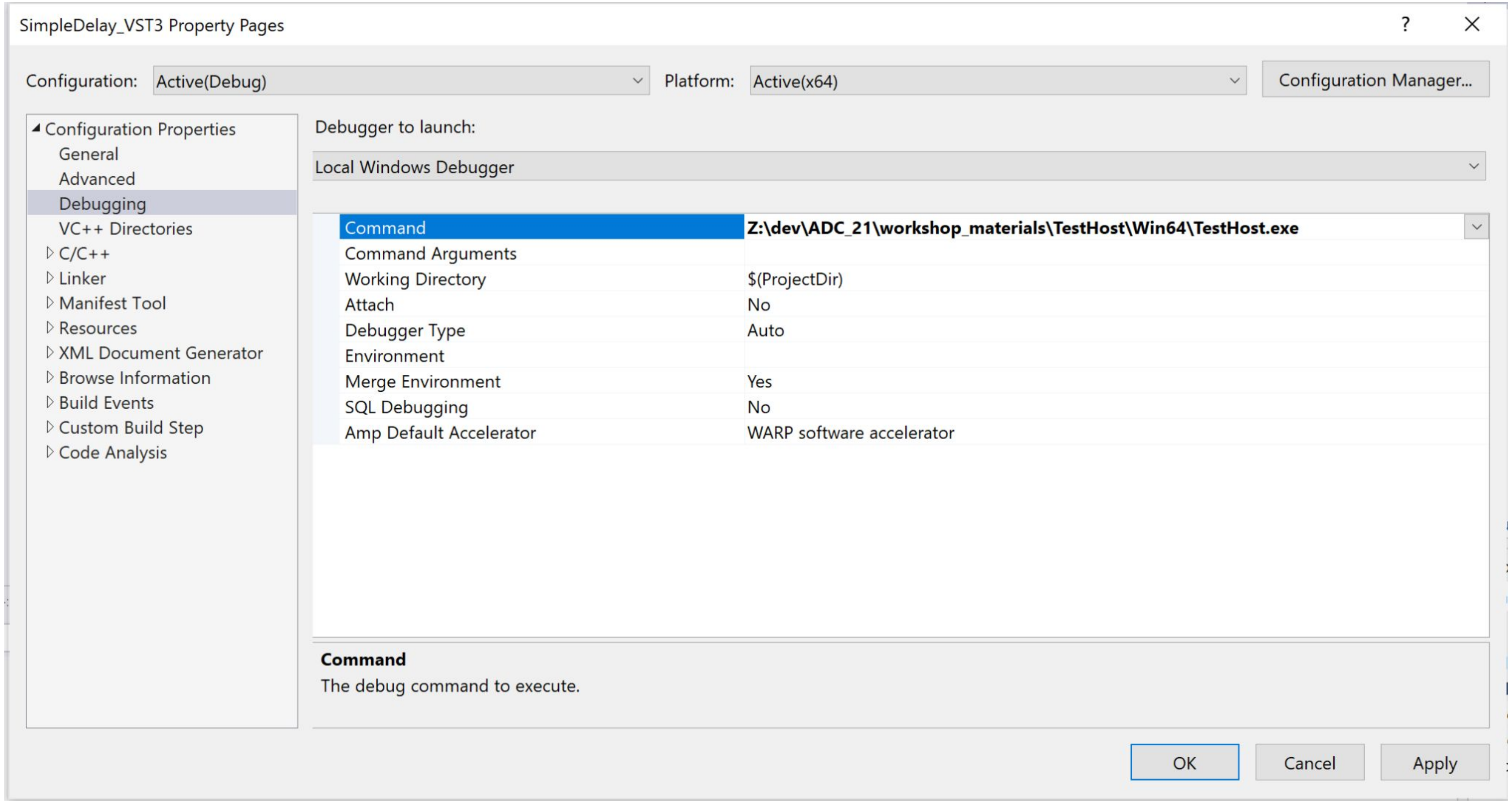
dev › ADC_21 › workshop_materials › TestHost › Win64

Name	Date modified	Type	Size
.DS_Store	02/11/2021 12:40	DS_STORE File	7
TestHost.exe	02/11/2021 13:02	Application	5,450

File name: TestHost.exe

All Files (*.*)

Windows



SimpleDelay_VST3 Property Pages

Configuration: Active(Debug)

Platform: Active(x64)

Debugger to launch:

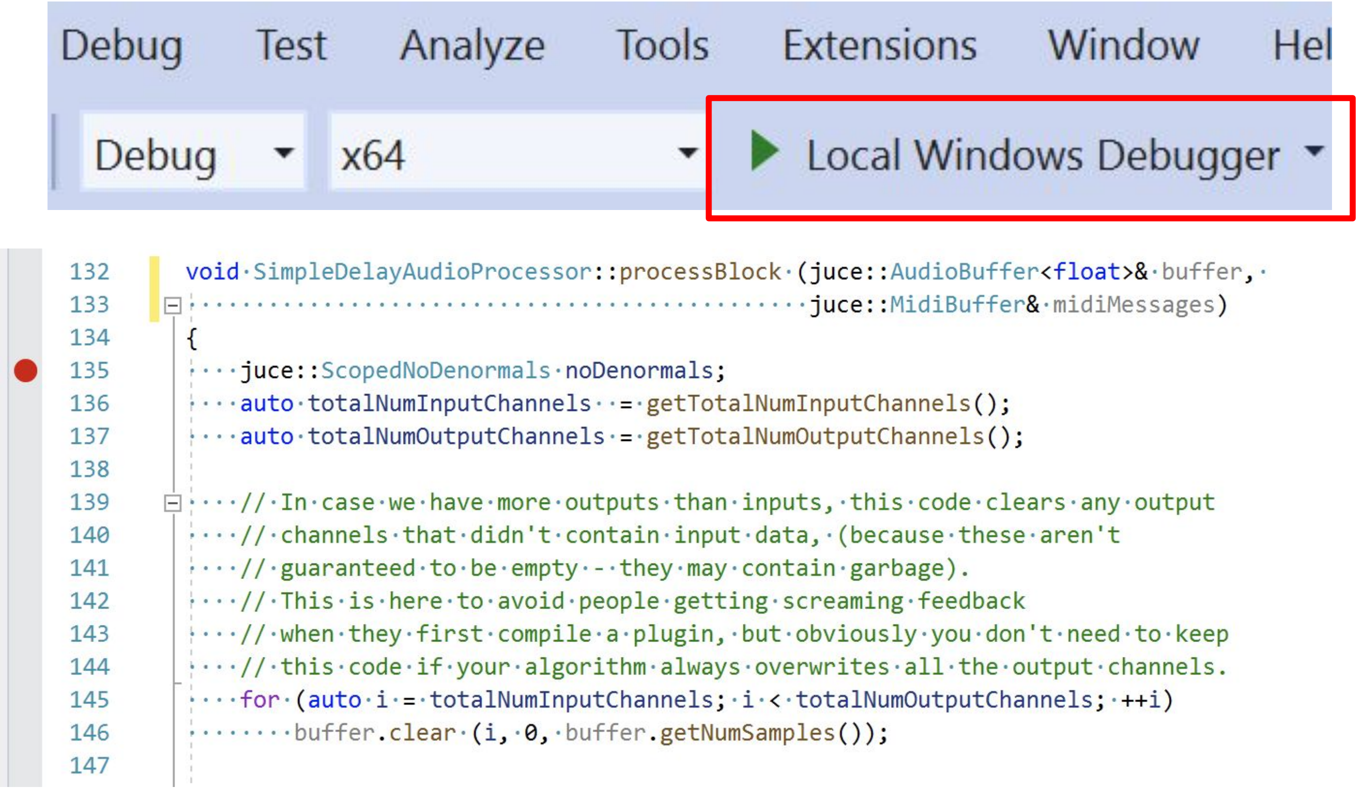
Local Windows Debugger

Command	**Z:\dev\ADC_21\workshop_materials\TestHost\Win64\TestHost.exe**
Command Arguments	
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command

The debug command to execute.

Windows

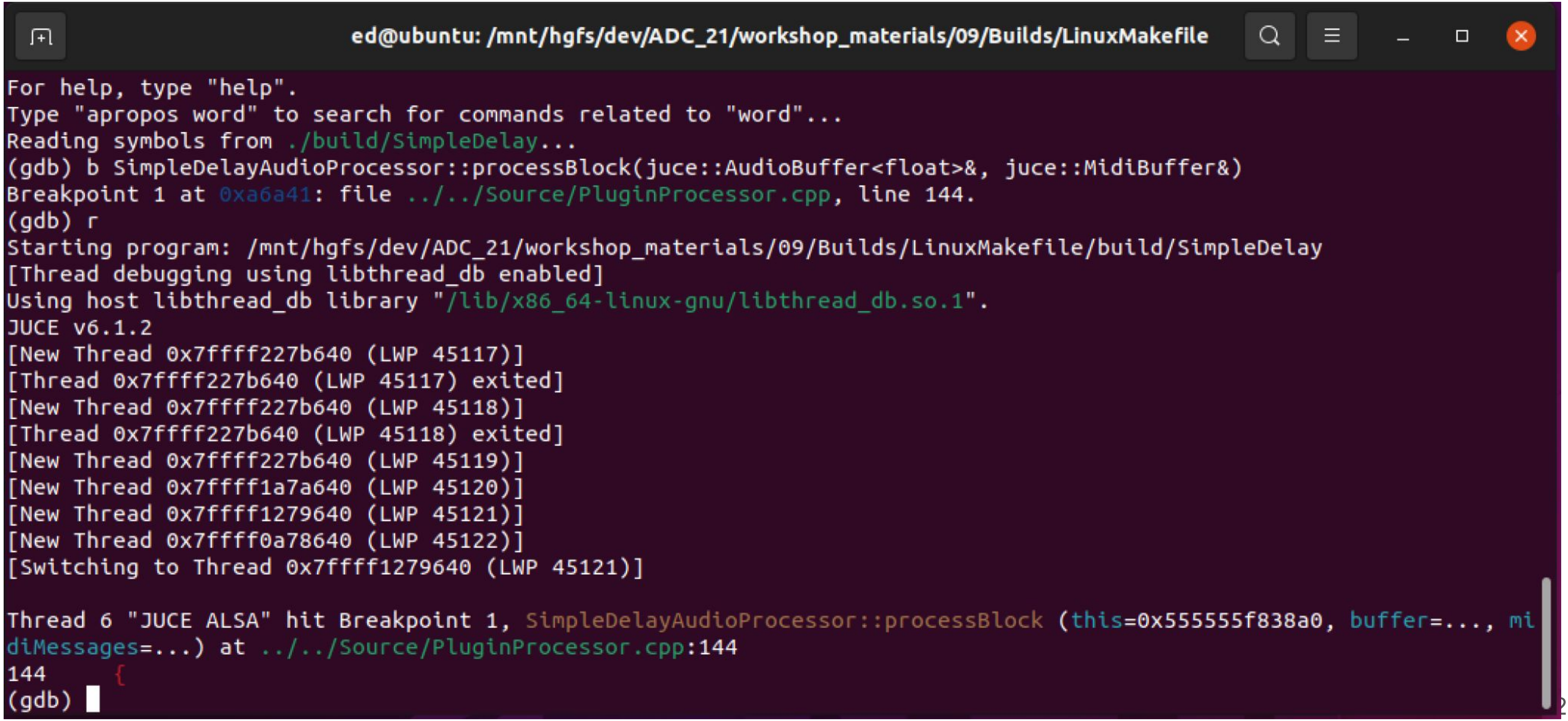


```
void SimpleDelayAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                              juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels  = getTotalNumInputChannels();
    auto totalNumOutputChannels = 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());

```

Linux

```
cd workspace/Builds/LinuxMakefile

make CONFIG=Debug

gdb ./TestHost/Linux/TestHost

break SimpleDelayAudioProcessor::processBlock

run
```

Linux


ed@ubuntu: /mnt/hgfs/dev/ADC_21/workshop_materials/09/Builds/LinuxMakefile

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./build/SimpleDelay...
(gdb) b SimpleDelayAudioProcessor::processBlock(juce::AudioBuffer<float>&, juce::MidiBuffer&)
Breakpoint 1 at 0xa6a41: file ../../Source/PluginProcessor.cpp, line 144.
(gdb) r
Starting program: /mnt/hgfs/dev/ADC_21/workshop_materials/09/Builds/LinuxMakefile/build/SimpleDelay
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
JUCE v6.1.2
[New Thread 0x7ffff227b640 (LWP 45117)]
[Thread 0x7ffff227b640 (LWP 45117) exited]
[New Thread 0x7ffff227b640 (LWP 45118)]
[Thread 0x7ffff227b640 (LWP 45118) exited]
[New Thread 0x7ffff227b640 (LWP 45119)]
[New Thread 0x7ffff1a7a640 (LWP 45120)]
[New Thread 0x7ffff1279640 (LWP 45121)]
[New Thread 0x7ffff0a78640 (LWP 45122)]
[Switching to Thread 0x7ffff1279640 (LWP 45121)]

Thread 6 "JUCE ALSA" hit Breakpoint 1, SimpleDelayAudioProcessor::processBlock (this=0x555555f838a0, buffer=..., mi
diMessages=...) at ../../Source/PluginProcessor.cpp:144
144     {
(gdb) 
```

Out-of-process loading

- Some hosts load plug-ins in a separate process
- Bitwig, Reaper (with some settings), AUv3s

- Need to attach to the plug-in process not the host process:
  - Xcode: Debug->Attach to Process by PID or Name…
  - Visual Studio: Debug->Attach to Process…
  - GDB: `attach <PID>`

macOS Notarised Hosts

- Since macOS Catalina (10.15), apps distributed outside the App Store must be notarised
- Apps can only be debugged with if they have the `com.apple.security.get-task-allow` entitlement set to `true`
- Must be `false` for notarisation to succeed
- However it is possible to re-sign host binaries!

macOS Notarised Hosts

- Get app’s entitlements using:

```
codesign -d --entitlements :- /path/to/host.app
```

- Modify them to include

```
<key>com.apple.security.get-task-allow</key>
    <true/>
```

- Set the new entitlements using:

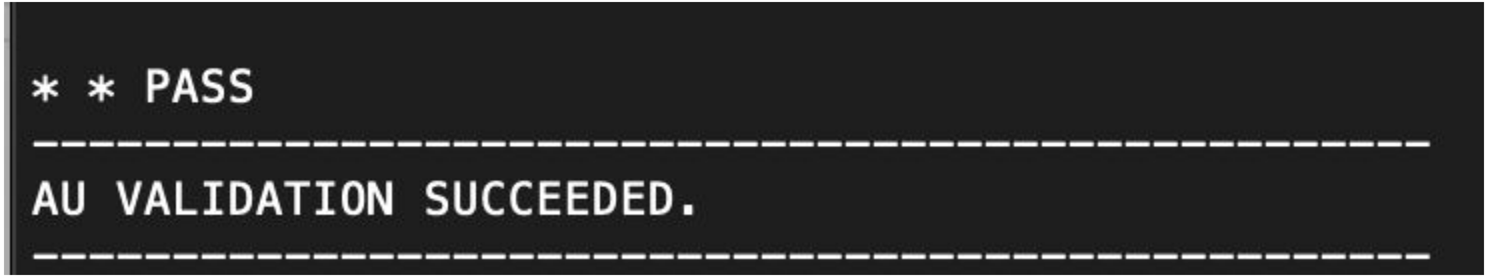
```
codesign --force --options runtime --sign - --entitlements
/path/to/plist "/path/to/app"
```

TESTING WITH TOOLS

auval

- Apple’s command line AU verification tool
- Tests basic AudioUnit functionality

```
auval -v aufx DLAY JUCE
```



```
* * PASS
--------------------------------------------------
AU VALIDATION SUCCEEDED.
--------------------------------------------------
```

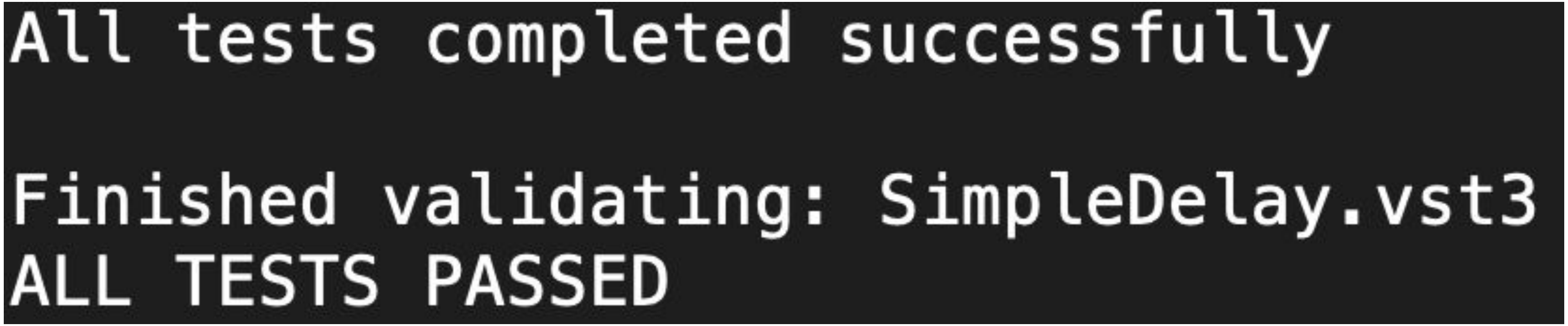
pluginval

- Cross-platform plug-in validation tool for testing AU/VST/VST3
- https://github.com/Tracktion/pluginval for source code and tagged releases
- Can be run on the command line or with a GUI

Audio
Developer
Conference

pluginval

```
pluginval --strictness-level 10 --validate SimpleDelay.vst3
```



```
All tests completed successfully

Finished validating: SimpleDelay.vst3
ALL TESTS PASSED
```

Possible plug-in improvements

- Parameter change smoothing
- A variable delay length
- Fractional delay lengths
- A `LookAndFeel` to style the widgets

https://github.com/juce-framework/JUCE

https://juce.com/

https://juce.com/learn/tutorials

https://twitter.com/JUCElibrary