Coding workshop: Examining the JUCE application

How the application starts up - follow the macros!

In this section, we'll work through the codebase to find out exactly how a JUCE application is created and started up.

The entry point for the app is in Main.cpp. There is no classic main function, but there is a call to a macro here:

```
START_JUCE_APPLICATION (OtoDecksApplication)
```

START_JUCE_APPLICATION is the name of the macro and OtoDecksApplication is the name of the class defined in Main.cpp and is the main wrapper class for your application. It might be DJApplication in your code.

The macro eventually generates a main function for us. If you right-click on the macro and follow the definition, you will get to another macro called JUCE_MAIN_FUNCTION_DEFINITION. That then calls JUCE_MAIN_FUNCTION, which generates the first line of main.

JUCE_MAIN_FUNCTION_DEFINITION continues and defines some code for the body of the main function, ending with return juce::JUCEApplicationBase::main (JUCE_MAIN_FUNCTION_ARGS);, where the args are 'OtoDecksApplication', or whatever your main application class is called.

If we look at the definition for juce::JUCEApplicationBase::main, we can see that it sets up an instance of the required class with the createInstance function, then sets up a message dispatcher.

Eventually, the initialise function for our application is called, and this calls the MainWindow constructor (both in Main.cpp). MainWindow creates an instance of MainComponent:

```
setContentOwned (new MainComponent(), true);
```

Now we are into MainComponent.h and MainComponent.cpp.

Experiment with macros yourself

You can experiment with macros yourself. Create a new console application in your IDE. Put this into your main CPP file, compile it and run it:

```
#include <iostream>
#define HELLO_WORLD  std::cout << "I am from a macro! " );

int main()
{
```

```
   HELLO_WORLD
}
```

Multiline macro

```
#include <iostream>
#define HELLO_WORLD  \
std::cout << "I am from a macro! " \
<< std::endl;

int main()
{
   HELLO_WORLD
}
```

MainComponent.h and cpp

MainComponent is where we'll do our first programming. In fact, you have already added a TextButton field to this class.

In MainComponent.h, we can see the following line, assuming you created an Audio Application:

```
class MainComponent   : public AudioAppComponent
```

This tells us that MainComponent inherits from AudioAppComponent. Inheritors of AudioAppComponent have to implement certain things:

- They have to generate audio
- They have to act like GUI components.

The function members of the MainComponent class

Here are the functions, constructors and destructors in the class definition:

```
MainComponent();
~MainComponent();

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

void paint (Graphics& g) override;
void resized() override;
```

Audio-related function members

prepareToPlay, getNextAudioBlock and releaseResources all relate to audio. prepareToPlay is called first, once. Then getNextAudioBlock will be called repeatedly until the application quits. If you have learned Processing, p5 or openframeworks, getNextAudioBlock is the audio equivalent of the draw function - here you will write the code that generates all the audio. releaseResources is called when the application exits and you should use it to free up any audio resources.

Experiment with the audio-related function members

Try putting in print out statements to the audio functions. How often is getNextAudioBlock called?

We will look at the audio-related functions in more detail in a later worksheet.

Drawing related function members: paint

paint is called whenever the application needs to be drawn. Try putting a print out into the paint function, running the application and seeing when paint is called. For example, is it called when you move another application over the top of your application?

You will see this line in the paint function:

```
g.fillAll (getLookAndFeel().findColour (ResizableWindow::backgroundColourId));
```

g is a Graphics object. Think of it like a wrapper around the current window's graphical content which provides functions allowing you to draw into the window.

fillAll fills the whole window (or strictly all the space taken up by the component represented by this Graphics object) with a colour.

To get a colour, we access the application's lookAndFeel and ask it what the background colour is. This keeps the colour palette of the application consistent. But it makes things more complicated.

We could also pass a colour we have defined ourselves like this:

```
g.fillAll( Colour{255, 0, 0});
```

Can you guess what colour that will be? Note that JUCE uses the British English spelling of colour, unlike many libraries you will encounter.

Drawing related function members: resized

resized is called whenever the application window changes size. It is also called once when the application window first opens. You'll see the code you wrote before that sets the size of the button here. You need to set the size of any GUI widgets you add to your application in the resized function if you want to see them.