Building Graphical User Interfaces with JUCE

Sean Enderby

Birmingham City University

14th December 2016

JUCE User Interfaces

The Component Class

- JUCE GUIs are built from instances of the *Component* class.
- The interface as a whole is define by a class which inherits *Component*.

```
class MyDankInterface : public Component
{
        // Implementation details.
};
```

- Other *Components* can then be added to the main interface *Component* as its children.
- JUCE provides pre-written *Component* classes for all the usual GUI widgets (Buttons, Sliders, Text Boxes...).

Component Appearance

- A *Component* defines its appearance in its *paint()* function.
- *paint()* is passed a reference to a *Graphics* object onto which the component should draw itself.
- The *Graphics* class provides functions for drawing shapes, images and pretty much anything you can imagine.

```
void MyDankInterface::paint (Graphics &g)
{
    // Fill the entire component with red.
    // JUCE provieds a lot of predefined
    // colours as static members of the
    // Colours class.
    g.fillAll (Colours::red);
}
```

Component Size and Shape

- All *Components* are rectangular in shape.
- More complex shapes are achieved by making certain portions of the rectangle transparent.
- A *Component's* width and height are defined in units of pixels.
- A *Component's* size can be set using its *setSize()* function.

```
MyDankInterface::MyDankInterface()
{
    // Make the interface 200 pixels wide
    // and 100 pixels high.
    setSize (200, 100);
}
```

Component Coordinates

- Positions within a *Component* are represented by x and y coordinates in units of pixels.
- (0, 0) represents the top left corner of the *Component*.
- x values increase from left to right.
- y values increase from top to bottom.

```
void MyDankInterface::paint (Graphics &g)
{
  // Draw a blue rectangle
  // 20 pixels wide and 15 pixels high
  // with its top left corner at position (10, 30)
  g.setColour (Colours::blue);
  g.fillRect (10, 30, 20, 15);
}
```

Child Components

- Child *Components* can be added to a *Component* using its *addAndMakeVisible()* function.
- The positions and sizes of these children can then be set using their *setBounds()* functions.

```
class MyDankInterface : public Component
{
public:
    MyDankInterface()
    {
        // Add the slider, position it at (10, 10)
        // and set its size to 180 by 20 pixels.
        addAndMakeVisible (slider);
        slider.setBounds (10, 10, 180, 20);
    }

private:
    // The slider is a member of our interface class.
    Slider slider;
};
```

Component Listeners

- Generally we want things to happen when a user interacts with *Components* in our interface.
- To do this we can attach listeners to *Components*.
- A *Component* will notify all its listeners when it has been interacted with.
- The GUI widget *Components* within JUCE all provide listener base classes (e.g. *Slider* provides *Slider::Listener*).

Component Listeners Cont.

- In order for our interface class to react to changes in its child *Components* it should inherit their listener classes and implement the relevant functions.
- The interface class can then 'listen' to other *Components* by adding itself to their listener list using their *addListener()* function.
- The *Slider::Listener* class provides the *sliderValueChanged()* function which will be called when *Sliders* being 'listened' to are moved.

Component Listeners Cont.

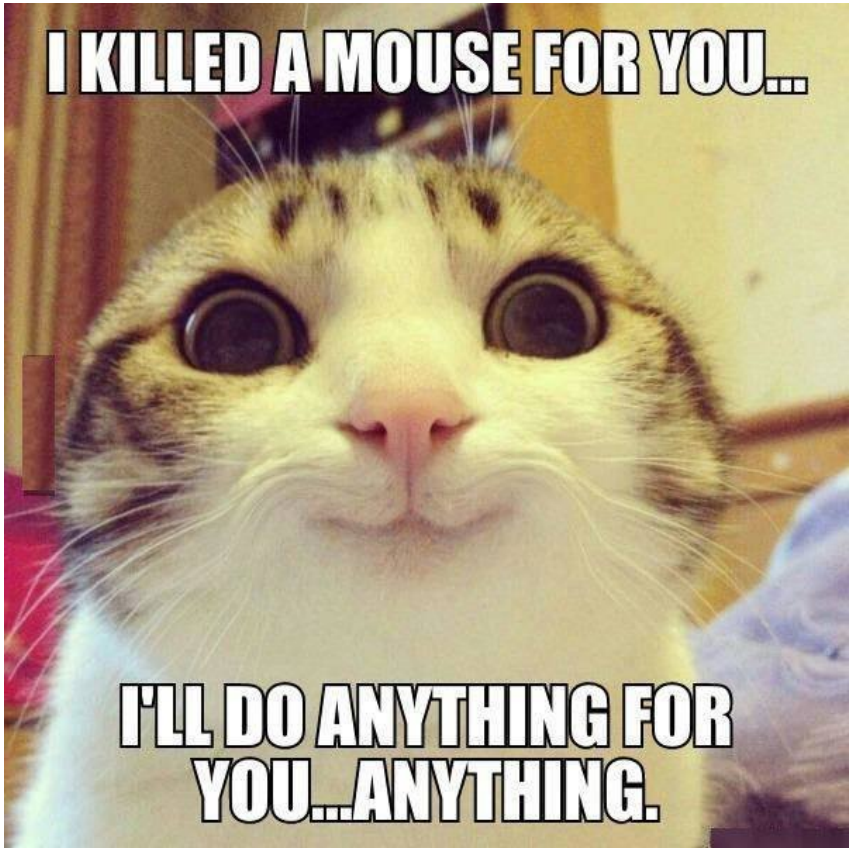
```
// Inherit both Component and Slider::Listener
class MyDankInterface : public Component,
                        public Slider::Listener
{
public:
    MyDankInterface()
    {
        // Add the slider and position it.
        addAndMakeVisible (slider);
        slider.setBounds (10, 10, 180, 20);

        // Start 'listening' to the slider.
        slider.addListener (this);
    }

    void sliderValueChanged (Slider *slider) override
    {
        // Do whatever we want to do when
        // a slider the interface is 'listening' to is moved.
    }

private:
    // The slider is a member of our interface class.
    Slider slider;
};
```

JUCE Plug-In Editors



The AudioProcessorEditor Class

- An audio plug-in's interface is know as its editor.
- JUCE provides a base class (*AudioProcessorEditor*) for making plug-in editors.
- *AudioProcessorEditor* inherits *Component* so our editor will have all the functionality of a *Component*.
- *AudioProcessorEditor's* constructor requires a pointer to the plug-in (*AudioProcessor*) object it belongs to as its argument.

```
class MyDankEditor : public AudioProcessorEditor
{
public:
    MyDankEditor (MyDankAudioProcessor &owner)
        // Base class constructors are called in the initialisation list.
        : AudioProcessorEditor (&owner),
    {
    }
};
```

Creating an Editor Object

- Our plug-in needs to be able to make a new editor and give it to the host on request.
- Remember when we disabled this functionality last week.

```
// We need to include the declaration of our editor class.
#include "PluginEditor.h"

bool MyDankAudioProcessor::hasEditor() const
{
    // Inform the host we do provide our own editor.
    return true;
}

AudioProcessorEditor* MyDankAudioProcessor::createEditor()
{
    // Create an editor and pass in this instance of our plug-in class.
    return new MyDankEditor (*this);
}
```

Editor Functionality

- A plug-in editor is responsible for three things.
- Defining what it looks like.
  - This is done in its *paint()* function and through its child *Components*.
- Controlling the parameters of the plug-in.
  - This can be done by holding a reference to the plug-in object.
- Updating itself if the plug-in's parameters are changed by the host.
  - This can be achieved using the *Timer* class in JUCE which allows you to define a function which will be called periodically.

A Full Editor Class

- A full plug-in editor class might look like this.

```
class MyDankEditor : public AudioProcessorEditor,
                     public Slider::Listener, // Allows us to listen to sliders.
                     private Timer // Allows us to call a function periodically.
{
public:
    MyDankEditor (MyDankAudioProcessor &owner);

    // Called to draw our editor.
    void paint (Graphics &g);

    // Called when sliders are moved.
    void sliderValueChanged (Slider *s) override;

    // Called periodically to update the slider positions.
    void timerCallback() override;

private:
    MyDankAudioProcessor &processor; // Hold a reference to the plug-in.
    Slider slider;
};
```

Editor Contructor

- In our editor's constructor we can add the child components and set their properties.

```
MyDankEditor::MyDankEditor (MyDankAudioProcessor &owner)
    : AudioProcessorEditor (&owner),
      processor (owner) // Initialise the processor variable.
{
    // Set the editor's size.
    setSize (200, 100);

    // Add the slider, set its position and range and add a listener to it.
    addAndMakeVisible (slider);
    slider.setBounds (10, 10, 180, 20);
    slider.setRange (-20.0, 20.0, 0.0);
    slider.addListener (this);

    // Start a timer so we can update the slider if the
    // plug-in's parameters are changed by the host
    startTimerHz (25);
}
```

Controlling Plug-In Parameters

- We can use the *sliderValueChanged()* function we inherited from *Slider::Listener* to change the plug-ins parameters when our sliders are moved.

```
void MyDankEditor::sliderValueChanged (Slider *s)
{
    // Check which slider moved and
    // update the relevant plug-in parameter.
    if (s == &slider)
        *processor.param = slider.getValue();
}
```

Updating the GUI

- We can use the *timerCallback()* function we inherited from *Timer* to update the sliders position periodically.

```
void MyDankEditor::timerCallback()
{
    // Set the value of the slider without
    // notifying its listeners. This way we
    // avoid the update looping forever.
    slider.setValue (*processor.param,
                     dontSendNotification);
}
```

Thanks For Listening!

Any Questions?

Let's Make a Dank Editor!