7.301: Implementing event listeners in JUCE

Introduction

In this worksheet, we will implement some event listeners in our JUCE application so that we can respond to users pressing buttons and moving sliders on the interface. Along the way, we will encounter pure virtual functions.

About event listeners

Events are things that happen while the program is running. One thing that can happen is that the user might click on a button in the user interface. We might say that the Button emits events. An event listener is a piece of the program (in our case, an object) which knows what to do when certain types of events occur. Event listeners need to register with the event emitter to receive events.

The Button::Listener class and pure virtual functions

Load up your JUCE application code in your IDE. In the MainComponent.h file, adjust the class declaration, so it reads like this:

```
class MainComponent   : public AudioAppComponent,
                        public Button::Listener
```

This says that MainComponent inherits from both AudioAppComponent and Button::Listener.

Therefore we can say that the MainComponent class is *polymorphic* - it can behave like an AudioAppComponent object or it can behave like a Button::Listener object. It has many (poly) forms (morphic).

C++ allows classes to enforce inheriting classes to implement certain functions. The Button::Listener class can say - if you want to be able to morph into a Button::Listener, you have to implement this function. We call this technique *pure virtual functions*. In JUCE, the developers use pure virtual functions to force you to implement necessary functions.

Look up the documentation for Button::Listener on the JUCE website. You should see that there is one function which we are forced to implement:

```
virtual void    buttonClicked (Button *)=0
```

We know we are forced to implement it because it has =0 at the end of its signature. This means that the Button::Listener class does not have any implementation for this function, so if you inherit from it, you must provide one or face the dreaded linker error when the linker tries to find an implementation can not.

Effectively, this means classes that inherit from Button::Listener must provide a buttonClicked function.

You can read more about pure virtual functions in textbook Chapter 14, p562.

Implement the buttonClicked function

Let's do the work to provide an implementation of buttonClicked. Add this to MainComponent.h - in the public section of the class:

```
void buttonClicked(Button* button) override;
```

Then add this implementation to MainComponent.cpp:

```
void MainComponent::buttonClicked(Button* button)
{
    DBG("  MainComponent::buttonClicked: They clicked the button" );
}
```

Notes: * I prefix the printed out message with the name of the class and function. This makes it clear where the message comes from. * We added the override keyword to make it explicit that this is an overridden function from the parent class. * The function receives a pointer to a button. We will use this to figure out which Button they clicked * Why do you think that the incoming argument is type Button, not type TextButton?

Run the code - does it print out the message? Why not?

Register to receive events from the Button

The reason the message is not printed out is that we did not register for events from the Button. We do this by adding the following line to the constructor of the MainComponent, just after we add the Button to the interface with addAndMakeVisible:

```
playButton.addListener(this);
```

Note that 'this' is a pointer to the current object we are operating inside. This is the MainComponent object since we are in the MainComponent constructor. Let's check the signature for the addListener function on the TextButton class to see what it expects to be sent:

```
void addListener (Listener *newListener)
```

So it requires a Listener. But we are sending it a MainComponent, right? No MainComponent can also morph into a Button::Listener, remember? But a Button::Listener is not a Listener, right? Wrong again. addListener is namespaced into the Button class, so Listener actually means Button::Listener.

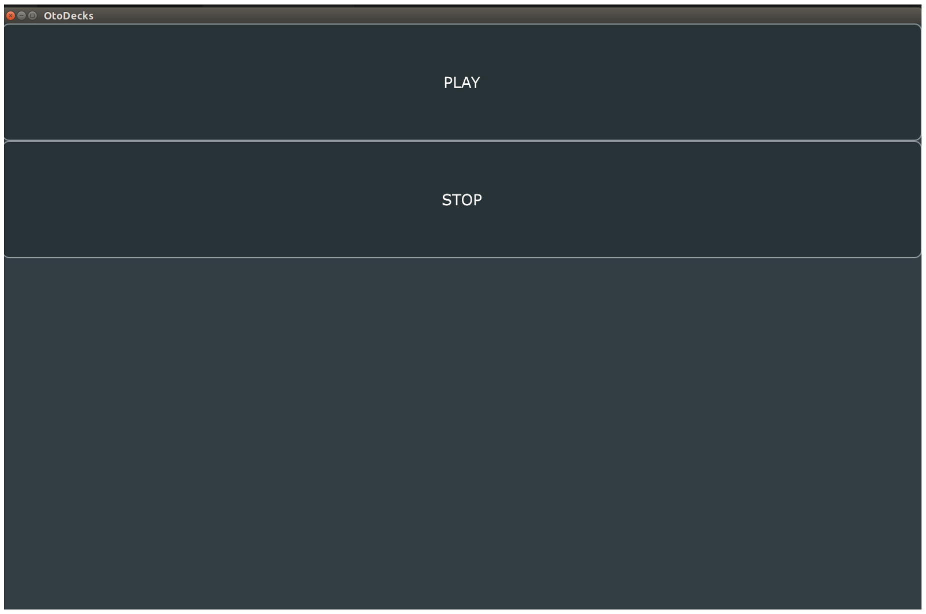
After all that, it is time to test the program. The program should print out the message when you click the button.

Add another button and register for events from that

Now go ahead and add another button. Carry out these steps:

- Add a new TextButton variable to the MainComponent.h. Call it stop-Button
- Call addAndMakeVisible with the Button in the MainComponent constructor
- Call addListener on the new Button
- Call setButtonText on the playButton and the stopButton variables, setting their texts to PLAY and STOP. Look up the function in the JUCE documentatiom if you are not sure how to use it.
- Finally, in resized, work out how to place and size the stop button, so it appears below the play button.

You should end up with something like this:



Figuring out which Button they clicked

The problem is, we get the same message regardless of which Button they pressed. We would like to run different code, depending on the Button. We need to use the Button argument sent into MainComponent::buttonClicked. Here is the signature of that function:

```
void MainComponent::buttonClicked(Button* button);
```

The Button argument is a pointer. Pointers contain memory addresses. A GUI object is really just some data stored in memory, and the two buttons will be stored in different places in the memory and will have different addresses. How might we use this idea to differentiate between the buttons? Simple compare memory addresses:

```
void MainComponent::buttonClicked(Button* button)
{
    DBG("  MainComponent::buttonClicked: They clicked a button" );
    if (button == &playButton ) // clicked button has same memory address as playButton
    {
        DBG("  MainComponent::buttonClicked: playButton" );
    }
    if (button == &stopButton )
    {
        DBG("  MainComponent::buttonClicked: stopButton" );
    }
}
```

You can read the comparison 'button == &playButton' as 'if the memory address stored in the button pointer is the same as the memory address of the playButton variable, then they are the same Button.

Verify that the two buttons are detected correctly. You should see some output in your console like this when you click the two buttons:

```
MainComponent::buttonClicked: They clicked a button
MainComponent::buttonClicked: playButton
MainComponent::buttonClicked: They clicked a button
MainComponent::buttonClicked: stopButton
```

Add a slider

Now we are going to add a slider to the interface to control the volume. We do not have any sound to control yet, but we will get to that later.

Back to MainComponent.h, in the private section of the class:

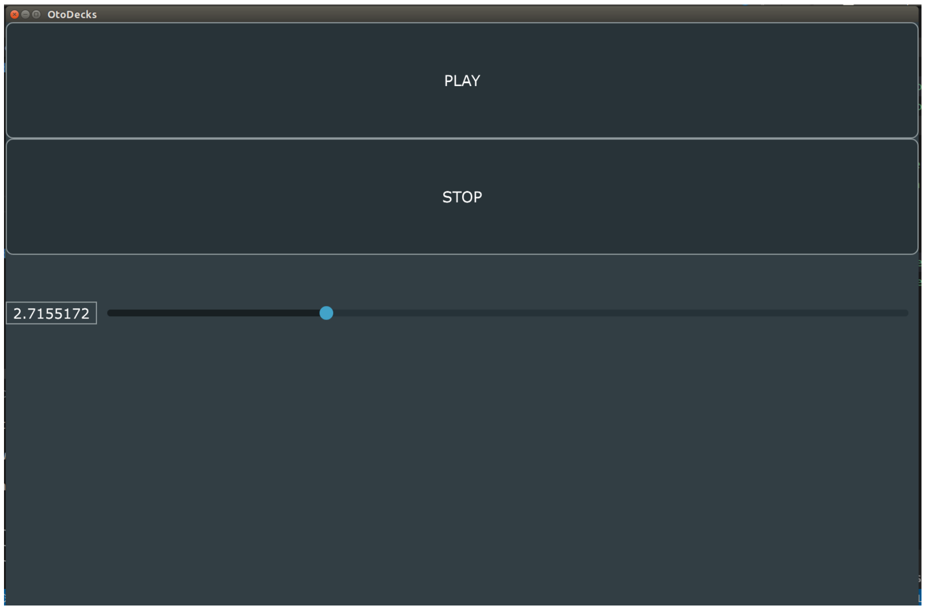
```
juce::Slider gainSlider;
```

(It is not necessary to prefix with juce, but it makes it explicit where that class comes from and is not too verbose.)

Now carry out the following steps, based on your experience with the buttons:

- Call addAndMakeVisible with the slider in the MainComponent constructor
- In resized, work out how to place and size the slider, so it appears below the stop button.

You should end up seeing something like this:



Receiving events from the slider

The final step is to implement another interface so that we can receive events from the slider. To receive events from a slider, we need to implement the Slider::Listener interface. Look it up in the JUCE docs and try to identify the function we are forced to implement.

That's right:

```
virtual void    sliderValueChanged (Slider *slider)=0
```

The give away is the =0 at the end, as with the button listener. Go ahead and add the interface to your class definition:

```
class MainComponent   : public AudioAppComponent,
                        public Button::Listener,
                        public Slider::Listener
```

Then add the necessary function to the public section of MainComponent.h:

```
void sliderValueChanged (Slider *slider) override;
```

Then the implementation in MainComponent.cpp:

```
void MainComponent::sliderValueChanged(Slider* slider)
{
    if (slider == &gainSlider)
    {
        DBG("MainComponent::sliderValueChanged: gainSlider" );
    }
}
```

Finally, register as a listener to slider events:

```
gainSlider.addListener(this);
```

You should notice that the pattern is very similar to the TextButton.

Accessing the value from the slider

When the user clicks the buttons, we do not need to know much aside from which Button they clicked. When they move the slider, we want to know what the value is on the slider. We can do that as follows:

```
void MainComponent::sliderValueChanged(Slider* slider)
{
    if (slider == &gainSlider)
    {
        DBG("MainComponent::sliderValueChanged: gainSlider " << gainSlider.getValue() );
    }
}
```

If you have everything set up correctly, you should see output in the terminal like this as you move the slider:

```
MainComponent::sliderValueChanged: gainSlider 0.711207
MainComponent::sliderValueChanged: gainSlider 0.704023
MainComponent::sliderValueChanged: gainSlider 0.696839
MainComponent::sliderValueChanged: gainSlider 0.689655
MainComponent::sliderValueChanged: gainSlider 0.682471
MainComponent::sliderValueChanged: gainSlider 0.675287
```

Here are some things to experiment with:

- Try adding some other types of components. Can you implement listeners for those?
- Try changing the style of the slider with setSliderStyle

Conclusion

We have seen how we can add button event listeners to our JUCE application that can differentiate between different buttons. We have also seen another event listener pattern for the slider component.