Juce Programming Tutorial

Table of Contents

Introduction

Most programming tutorials will start out by teaching you some fundamental (and fundamentally boring) core code things. For example, when you were learning C++, you probably did a whole bunch of boring processes on integers, floats, strings, functions, pointers, etc. You were able to see what you were doing by using the ready defined 'cout' output stream, stuffing your results out onto the console. Yes, the console. My own progression as a C++ programmer began in this way, and I remained at the console output stage because every other kind of graphical output code was way to frightening to contemplate. I considered getting into Windows application development, but those darned Microsoft classes and typedefs made me nearly wet my pants. Yes, I mean that in a bad way.

Everyone wants to be able to get into programming something that displays nicely on the screen with graphical elements, but it's always such a chore to get to that point that it's easy to burn out your enthusiasm. I'll be honest – it happened to me. I stopped programming for quite some time.

Then I discovered Juce. And again I nearly wet my pants. But this time, I mean that in a good way; all of a sudden I was able to 'do stuff' that looked *nice* and wasn't scary! It took me a while to get my head around it all though, but I can promise you that it's a far easier experience than any other libraries I've happened upon.

This tutorial is probably not very much like many other programming tutorials. I say that because I'm going to teach you things the other way round. We will start with the 'fun stuff', because I feel it's important to understand that first. This has two benefits. Firstly, it's quite exciting being able to build graphical applications right from the start of your experience. More importantly though, it gets you to a stage where you know how to display the results of your 'cogs and gears' behind the scenes experimentation.

Juce has many classes that you can use throughout your program. Pretty much every base is covered (Strings, Arrays, Files, etc) and you'll find you can make do without any other libraries. If I started by teaching you these though, you'd have to take a lot of stuff for granted when I give you the basic tools to put your results on the screen.

The aim of this tutorial then is to give you all the knowledge you need to be able to start programming your applications using the Juce classes. I won't cover all of the classes here, as many are very specialised. With luck, you should understand enough about how Juce does things by the end that you can easily figure out how to use these other classes by yourself, by reading the (very detailed) documentation. I will write additional guides on how to use certain Components though (for example, the ListBox, the TreeView, etc…).

With all that said, I think it's high time we began the tutorial!

Chapter 1 - The Starting Point

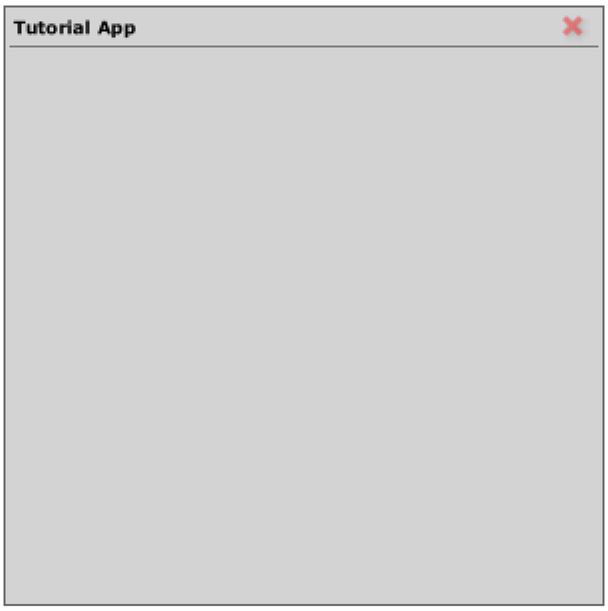
I'm not going to try to teach you how to build a whole Juce project from scratch. That would baffle you almost immediately. What I'm going to do is provide you with a set of 'starting point' files, and let you use them as a sort of 'sandbox' for learning the Juce way. These files are slightly modified versions of Jules' original demo app source files. They should be slightly easier to begin with.

I'm not going to attempt to explain how the 'application' code works, either. By the time you're in a position to worry about it, you'll surely know enough about Juce to figure it out for yourself. In fact, I've only recently bothered to look in detail at what it does and how it works. For the time being, you really can just take for granted that it does the hard stuff for you!

Create a new project in your development environment. Make sure that your paths (include, lib, etc) are all set correctly as indicated by the Juce documentation. In this new project, add the three StartingPoint tutorial files.

Provided you have Juce properly setup on your system (with the correct paths and dependencies all set appropriately), the tutorial files should compile without a problem. If you do have any problems, it means that either your project has some incorrect settings, or your environment does not have all the include/lib paths set properly. This is the time to discover such things! If you find yourself scratching your head, go to the forum and make some enquiries. You may spot what the error is yourself.

When you get the program compiled, you'll be massively disappointed with the result. The application should look like this:



Now I shall begin to use some Juce terminology. This app is a basic launch pad for your Juce programming, and it gives you a single DialogWindow, which holds a single Component. What the DialogWindow is should be quite obvious – that's the 'window' bit the application presents itself as. The Component is probably less clear. Don't worry though, we shall get on to that in the next section – suffice to say that it's probably the most important type you'll encounter!

Chapter 2 –Component Basics

A Juce application would be nothing without the splendid 'Component' class. If you want to display anything on the screen, a Component is what you need. All of the Juce GUI elements are subclassed from Component.

What is a Component?

A Component has the following crucial properties:

- It is a rectangular region that can be positioned and sized to your needs.
- It can be painted on, with text, images and vector based graphics.
- It can respond to mouse activity.
- It can hold other components in a specified layout.

A plain Component is just a clear rectangle with built in abilities as described above. You can set the size (width and height) and position (x,y) of the rectangle, and you can control its appearance by drawing directly onto it. It has 'callback' functions which are called when mouse activity happens over it. It can also be filled with any number of 'child' Components. The Component system is very sophisticated indeed!

How can I actually 'do stuff' with a Component?

We know that it can respond to mouse activity, and it can display any old thing we choose. To actually do this stuff though, we need to know a heck of a lot of Juce first, which currently (we shall assume) we do not. Thankfully, Juce comes with a big pile of ready-made Components which we can use to build up our application.

So far, we're aware that our tutorial application has a DialogWindow with a single Component for our content. In our code, this content Component is a custom class called

MainComponent, defined in MainComponent.h. Open this file and have a look at the code.

You’ll notice that not a lot is done with it. This explains why the application is so boring! We only have a constructor and a destructor. You’ll also spot the important bit – the class ‘is’ a Component, thanks to the inheritance declaration after the class name.

```
class MainComponent    :  public Component
```

This, as you should be aware from your masterful C++ knowledge, means that the class MainComponent has all the same members and behaviour as the Juce Component class. Thanks to the object oriented nature of C++, we don’t need to redefine the Component behaviour when we want to make our own version.

As we know from running the app, it’s just an empty Component. As we also know, it’s already aware of mouse activity. The important thing, though, is that – because it’s a Component – we can add other Components to it, which is what we shall do next.

How do I add a Component?

We’re going to use three ready made Component types here. These are the TextButton, the Slider, and the Label. It might be a good idea to consult the Juce documentation to get an idea of what these Components are. Basically, the TextButton is a nice general-purpose ‘button’. The Slider is just what you’d imagine (an adjustable slider for setting numerical values). The Label is a Component that displays text.

What do we need to know to be able to add these? Well, let's make some logical observations.

Firstly, if you’re to have a Component on your Component, you’re going to want to be able to 'use it' for stuff, so you'll to have access to it. You therefore need some kind of member variable to hold a pointer to it.

Secondly, if you want a Component to appear on your Component, it will need to be created, and it's a pretty safe bet to say that you want this to happen in your Component's constructor.

So, we have two simple things to take care of before we can add these Components.

1) Add a *private* pointer member (of the desired type) to our class.

This means, in the 'private:' section of our class declaration, we add an appropriate member…

Examples...

```
TextButton* button1;
Slider* slider1;
Label* label;
```

2) Create the desired Component in our Component's constructor.

In the constructor's function body, we need to instantiate these Components, taking care get any required parameters in their own constructor calls.

Examples...

```
button1 = new TextButton (T("Button 1"));
slider1 = new Slider (T("Slider 1"));
label = new Label (T("Label"), T("text here"));
```

Notice that, in the above examples, all the Component constructors have a 'String' type parameter. This will be covered later. These parameters set the 'name' of the Component (used for internal purposes where necessary). In the case of the Label, there is also a second parameter which specifies the initial text shown on the Label. For now, we're going to just put in literal strings, using the T("<string>") macro. This isn't strictly necessary, but it's handy to ensure that you're clearly defining a String literal.

We're going to add three TextButtons, a single Slider, and a single Label. That means the following lines:

In the private member declaration area...

```
TextButton* button1;
TextButton* button2;
TextButton* button3;
Slider* slider;
Label* label;
```

In the constructor...

```
button1 = new TextButton (T("Button 1"));
button2 = new TextButton (T("Button 2"));
button3 = new TextButton (T("Button 3"));
slider = new Slider (T("Slider"));
label = new Label (T("Label"), T("text here"));
```

Now, that's taken care of the 'existence' of these Components. They will be created inside the MainComponent, and will be accessible from within it. However, we still have yet to actually 'add' them to our Component! To do this, the Component class has a nice simple function for us to use. You simply call it with the pointer to the Component you wish to add, and you're done. The syntax is:

```
addAndMakeVisible (myComponent);
```

Of course, this will also need to happen in the constructor. Add the following lines after the Components have been created:

```
addAndMakeVisible (button1);
addAndMakeVisible (button2);
addAndMakeVisible (button3);
addAndMakeVisible (slider);
addAndMakeVisible (label);
```

That’s the Components added to our MainComponent! Exciting stuff! Sadly, though, we’re still not quite done. Firstly, before we forget, we need to remind ourselves that we’ve recently created these objects using the ‘new’ operator. This means they exist on the heap, and it is our responsibility to delete these and free the memory when the Component itself is destroyed. This might mean that we have to delete each one by hand in the destructor, but we must remember that they are registered with the Component.

Conveniently, because the Component class is a clever device indeed, there is a single function that we can call to ensure our toys are properly tidied away.

In our destructor, we just need to add the call:

```
deleteAllChildren ();
```

This will unregister all of the Components, and also delete them as it goes along. One call to rule them all. Splendid.

With that, we’re very nearly there. So far we’ve (1) added the child Components, and (2) made sure that they’re cleared away. If you’re in ‘sharp’ mode, you’ll perhaps have realised one crucial omission. The Component, while clever, is not magical enough to instinctively know how you want your controls arranged, so you have to do that yourself. This is very simple, and the easiest way to do this is by using the Component function ‘setBounds (…)’. A quick check of the docs shows that it has the following form:

```
Component::setBounds (x, y, width, height);
```

The first two parameters are the coordinates of the Component’s top-left corner. The other two specify the size of the Component. As you might expect, because this sets the layout of a particular Component, you call it for the Component you wish to place. So, to position ‘button1’, you’d put something like this:

```
button1->setBounds (10, 10, 50, 20);
```

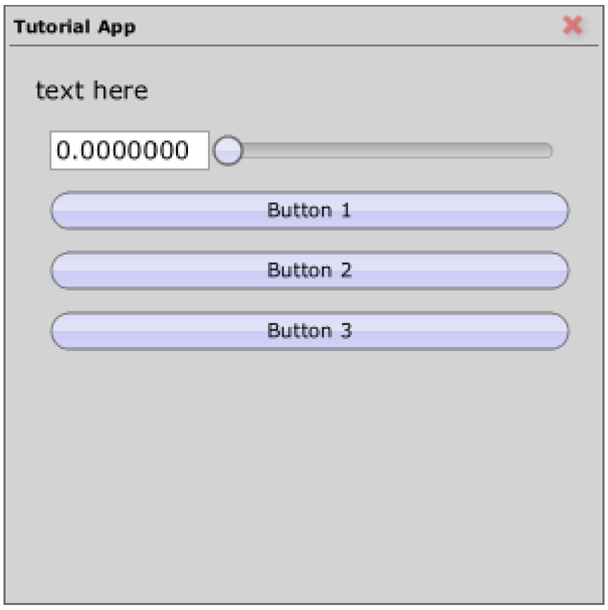
This would make it 50 pixels wide by 20 high, and would place it 10 pixels away from the top-left corner of the Component it lives within (in our case, MainComponent).

We need to make a call like this for each one of our child Components, and then we will be all set to compile and see our wonderful creation! It is possible to arrange these dynamically, based on the current size of the Component they reside in, but for now we shall keep things simple. If you look at the file MainHeader.h, you'll see some variables set to make the application configuration easy; two of these define the size of the app window – it is 300x300. This is actually the size of the DialogWindow, and the MainComponent sits within that (a little shorter due to the window's title bar, and there are a few pixels of padding around the edges). That's plenty of guidance though – we just make sure our controls don't go too wide or off the bottom of the window!

's make the following bounds calls (in the constructor, after the Components have all been created and added):

```
label->setBounds (10, 10, 280, 20);
slider->setBounds (20, 40, 260, 20);
button1->setBounds (20, 70, 260, 20);
button2->setBounds (20, 100, 260, 20);
button3->setBounds (20, 130, 260, 20);
```

Now we've positioned the Components, we are in a position to compile our program again! Build it, and you should see the app look like this:



Now we have ‘things’ on our window! You should hopefully be extraordinarily excited by this. However, as you’ll no doubt have noticed, these ‘things’ don’t really do anything. This is what we shall address in the next chapter, when we examine the message handling aspect of Juce.

Chapter 3 – Listening for Messages

The last chapter ended with us creating an app with three TextButtons, a Slider, and a Label. Simply by adding these Components to our MainComponent, we have a panel with real working widgets! The buttons highlight if you move the mouse over them, and if you click them they 'press' in. The slider can be dragged and its value changed. This is the power of Components! They have already been programmed to 'behave' in this way for us.

As we realised, though, these controls don't do anything other than look pretty. In a real application, we'd want something to happen when we click a button. Similarly, when we set a value on a slider, that change should be reflected in some other section, where the value will be used for whatever purpose.

Thankfully, we don't need to 'hack into' these controls to find out when they're used. They already broadcast messages to indicate their status. We just need to understand how these messages can be used so that we can 'do' this 'stuff' we want. This requires the introduction of message classes and concepts.

What are these message classes?

There are several types of message defined in Juce. For the time being, we shall ignore the actual types, and just focus on the fact that they use 'messages'. I shouldn't need to explain what a message is, as it is perfectly named; a message is something one object sends to another object to notify it that something has happened. In Juce, there are two categories of message-handling object you need to know about, and they are very simple. These are broadcasters and listeners.

Very simply, a broadcaster has the ability to send messages, and a listener is able to receive them and act upon them. That should be fairly intuitive. The only thing you need to be made aware of is that a listener needs to be registered with a broadcaster in order to be able to 'hear' these messages. That is the basic concept out of the way. In order to look closely at broadcasters, we'd need to know more than we do about Juce. This chapter will instead focus on *listening* to these messages. Let's look at an example, using the TextButton.

A TextButton is a "Button", thanks to the general purpose Button base-class (look at the docs and you'll notice there are various other button types, all based on Button). As mentioned before, Juce has a variety of different message types, and one of these is the button message. A Button is a *broadcaster*, generating button-related messages for your application to pick up.

The TextButton (being a Button) is therefore already configured for this; every time you click a button on our tutorial app, a button message is generated and sent out into the program. We've not yet done anything to make use of this though, so the message simply disappears.

What we need to do is create a ButtonListener (something that Juce comes with for us to use), which is specifically designed to respond to only button messages.

Where does the Listener go?

Our MainComponent is not just a rectangular panel with buttons and sliders on. It can have any behaviour we like. Because this Component 'owns' these controls, it makes sense that it should be 'in charge' of them, and that it should be the thing that responds to their messages. This is an important factor: your main Component is very likely to be the place where your actual 'program' code lives. It contains everything else, and so it is logical that it contains the behaviour to control and respond to everything else too.

Rather than trying to figure out where the Listener should *go*, it's better to think what the Listener would *be*. The Listener object is what responds to a message. We want our program to respond to the button messages. Our MainComponent *is* the program. Therefore, our MainComponent needs to *be* a ButtonListener.

You may be thinking "Hang on, our object is a Component, not a ButtonListener!", and you'd be correct. That doesn't mean that the two types are mutually exclusive. The powerful nature of Object Oriented Programming gives us the ability to say that an object is both of these things, and much more.

As we saw in the second chapter, we give an object the properties of an existing class by using inheritance (in the object's class declaration). In order to inherit multiple base classes, we simply add these classes in the same place, separated by a comma.

```
class MainComponent    : public Component,
                         public ButtonListener
```

This means our class is now a ButtonListener as well as a Component! Is that easy or what? Of course, being a listener isn't quite enough for an object to actually receive messages. As mentioned earlier, it needs to be registered with a broadcaster before it will actually get any messages. Also, while being registered is enough for the messages to find their way to the object, it still won't 'respond' unless you tell it what to do.

As you're hopefully aware, base classes can have virtual functions. This means there is an interface for a particular behaviour (in this case, a function that is called when a message is received), but the actual code is left for you to define in your subclass. The function in question for this example is as follows:

```
void buttonClicked (Button* button) = 0;
```

That's how it is declared in the ButtonListener base class. The '=0' at the end of the line means that it is a 'pure virtual' function. This means that you MUST define its behaviour for your class to be complete – if you don't, the compiler will tell you that you still have an abstract class, and will fail. The MainComponent is now a ButtonListener, so we must give it this function and tell it what to do.

In the public member section of MainComponent, add this code:

```
void buttonClicked (Button* button)
{
}
```

This is enough to satisfy the compiler, although it still won't actually do anything. When the object receives a button message, this function is called with a pointer to the button that generated the message (notice the parameter, Button*).

If you remember, though, we need to register a ButtonListener with a Button (which broadcasts button messages) before any messages will be received. This is very simple, and we do it using the following form:

```
myButton->addButtonListener (ButtonListener*)
```

The ButtonListener we want to register is MainComponent. The broadcasters we want to register it with are the three TextButtons we have created. We shall perform this registration in MainComponent's constructor. Because we are 'inside' MainComponent, we can get its pointer using the 'this' operator. Add these lines to the end of the constructor:

```
button1->addButtonListener (this);
button2->addButtonListener (this);
button3->addButtonListener (this);
```

This means that MainComponent will now receive messages whenever we click on any of the three TextButtons! It's that simple. If you click button1, it will generate a button message, causing MainComponent.buttonClicked () to be called with the address of button1 as the parameter.

Hopefully, you should already be able to see how we can respond to the button presses in our code. We have three buttons (button1, button2 and button3). The three member variables we created hold the addresses of these TextButton objects. If one is clicked, our buttonClicked function will receive one of these pointers. Thus, in order to determine which button is clicked, we need simply to compare the function's button parameter against our three button members:

```
void buttonClicked (Button* button)
{
   if (button == button1)
   {
      // respond to button1
   }
   else if (button == button2)
   {
      // respond to button2
   }
   else if (button == button3)
   {
      // respond to button3
   }
}
```

If we put some code in these sections, it will be executed when the corresponding button is clicked. That is all there is to it!

How can we demonstrate that this works? Well, we've already got a Label on our panel, which currently just shows "text here". We'll get it to change the text when we click on the buttons.

If you look at the Juce documentation for the Label class, you'll see that it has many functions. If you're an excitable and enthusiastic programmer, you'll probably have had a scan through several of the class docs. All the ready made Components have a wealth of specific functions to 'do stuff', which is something that we can of course do for our own Components. The function we're going to use is the appropriately named 'setText ()'. It has the following form:

```
Label::setText (const String& text, bool sendMessage);
```

The parameters may look a bit confusing at first, but we'll keep it simple for now. The 'sendMessage' parameter should indicate to you that this class is some kind of broadcaster. It is actually a ChangeBroadcaster, something which we'll cover later in this chapter. For now, we're going to ignore this fact, and just pass it 'false' so that it doesn't send a message. The first parameter (if you're not wholly used to C++ syntax) may look a bit scary, but the important thing is that it is a String. Again, as earlier, we're going to just use a string literal (and the T("<string>") macro) here, as Strings will be covered in a later chapter.

So, to set the text on this Label, we use the following line:

```
label->setText (T("Text of my choice"), false);
```

This will cause the Label to change its text to "Text of my choice". We want our TextButtons to change the text, so we put a line like this in the response section for each of our buttons in buttonClicked (), as follows:

```
void buttonClicked (Button* button)
{
   if (button == button1)
   {
      // respond to button1
      label->setText (T("Button 1 clicked!"), false);
   }
   else if (button == button2)
   {
      // respond to button2
      label->setText (T("Button 2 clicked!"), false);
   }
   else if (button == button3)
   {
      // respond to button3
```

```
        label->setText (T(“Button 2 clicked!”), false);
    }
}
```

If you compile this now, you’ll see the Label’s text change when you click the buttons. This gives you a prime opportunity to try out various exciting swear-words for each TextButton! Remember that the documentation lists many functions available for the Component types we’re using. See if you can carry out the following tasks:

Task

- Get each button to change the text on a different button.
- Get the buttons to set the value of the slider.

Consult the documentation for Slider and TextButton for clues about which functions you’d need to use.

What about the other message types?

We’ve now covered the basics of message handling, using the ButtonListener base class in our Component (and handling ‘Button’ messages). Now, we’ll quickly examine another type of message – the ‘Change’ message.

While we were working with the buttonClicked () callback function, we stumbled upon evidence that a Label is a ChangeBroadcaster. If you took the time to experiment in the previous section, you’ll have hopefully discovered for yourself that Slider::setValue () gives away a similar trait; a Slider is a ChangeBroadcaster too!

The Slider is the most obvious example of a ChangeBroadcaster. Whenever you adjust the control, it sends out a change message to any registered ChangeListeners. That means, if we want to be notified of any changes made on the Slider, we need our MainComponent to be a ChangeListener. Hopefully, that should have been fairly obvious to you!

In order to give our MainComponent this power, we do this:

```
class MainComponent    : public Component,
                         public ButtonListener,
                         public ChangeListener
```

We also have another pure virtual function to define, courtesy of the ChangeListener base class. This has a slightly different name, but works in a similar way to before. Add this to MainComponent:

```
void changeListenerCallback (void* changeSource)
{
}
```

This is called whenever a change message is sent from a ChangeBroadcaster we're registered with. Instead of a 'Button*' parameter, we have a 'void*', but in principle it works in exactly the same way; when this is called, a pointer to the source of the change is received. Thus, our callback will be given the pointer to the Slider. Well, it will, once we've registered with it!

In the constructor...

```
slider->addChangeListener (this);
```

This is nearly identical to the line required to register with a ButtonListener. Again, this gives us the ability to hear about changes made to the Slider, but we've still to put in any response code.

The first thing we'll need to do is check that it is indeed the Slider that has changed. This is a simple pointer check, as with the Button pointer check we did before.

Then, we'll do a similar thing to the TextButton response – just update the Label text, this time with the value from the Slider. This means that we need to get the value from the Slider, using the Slider::getValue () function. This function returns a *double* (high precision floating point number), so we'll create a variable to hold the result.

Remember, though, that the setText (…) function takes a *String* parameter for the text, whereas we wish to use a *double*. We shall therefore make our first genuine use of the String class, but only in passing for now. The String has a variety of different constructors, which can conveniently turn various different types into a String for us, behaving almost like a cast. Thus, in order for us to use a double as a String, we simply need to wrap it in a String constructor call.

```
void changeListenerCallback (void* changeSource)
{
   if (changeSource == slider)
   {
      double value = slider->getValue ();
      label->setText (String (value), false);
   }
}
```

With this function now added, a compile should show this program to work as expected. Now we are able to respond to both Button messages and Change messages without difficulty.

To briefly summarise what we've covered in this chapter:

- Broadcasters send messages, Listeners receive them via callbacks.
- Listeners must register with a Broadcaster to receive messages.
- Any class (e.g. Component) can inherit Listener behaviour.
- A Listener subclass must override the virtual callback function.
- The callback should check the source and respond accordingly.

We're still not quite ready to delve into the important world of the Juce core classes. There is one more aspect of Components we should understand before we can properly do things behind the scenes, and that is the paint () function. The next chapter will cover graphics.

Chapter 4 – Painting and Decorating

We touched on the fact that Components can be painted on in chapter 2. We’ve also seen hard evidence of this capability, because the TextButton, Slider and Label objects we’ve added (all of which are essentially Components) have been coded to draw shapes and text so that we can see them.

We’re now going to have a look at how we can make use of this feature ourselves. It’s actually quite simple, and by the end of this chapter, you'll have the tools to begin to design the appearance of your own widgets.

Where do I paint my Component?

The place this appearance design happens is in the ‘paint ()’ function of the Component. This is a virtual function, so we can override it customise the look of our Component. The function has the following form:

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

This function is a neatly prepared way for us to draw onto our Component; the single 'Graphics' parameter is the Component’s way of making our life easy. Notice that it is a reference, and that it is not 'const'; this shows us right away that it is something we can change. We are given an object, and we can do what we like to it.

Add the following code to the public section of MainComponent:

```
void paint (Graphics& g)
{
}
```

What is a 'Graphics'?

It might help to imagine a Graphics object as a special robot arm, whose sole purpose is to draw stuff onto a canvas. You can give it instructions on what to draw, how to draw it, and where on the canvas it should go, and it will happily oblige. A quick look at the documentation for the class will give you an idea of the kind of commands it can understand; names like 'drawLine()', 'drawText()' and 'fillAll()' are fairly self-explanatory.

In our paint function, we find ourselves presented with a 'Graphics'. This has already had its canvas assigned for us – that'd be the Component rectangle. If we tell it to draw anything, it will appear on our Component.

How big is my Component?

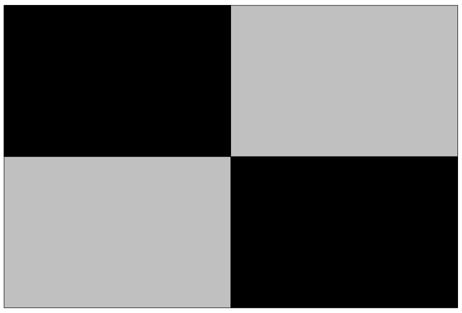
There are a few things you should know about the canvas you'll be drawing on. Firstly, it's actually called a 'context'! The next thing is the coordinate system. All the drawing operations take coordinates (that should be obvious – you need to specify *where* these things will get drawn), and these coordinates start from the top-left corner of the context. That means that bigger values of 'y' refer to a point further down the context, and bigger 'x' values are off to the right.

But hold on a moment! Right now, the only place we know the exact coordinates for is the top-left corner. How are we going to know how big to draw things? How do we know the size of the context, so that we don't go off the edges? For this important information, we turn to the Component itself.

The Component has had its bounds set (just like we did with the widgets in chapter 2), so we know it has a size. We can get this size with two simple calls – getWidth() and getHeight(). The paint() function is 'inside' the Component, so we have direct access to these functions. These simply return integers, which we can use to calculate coordinates that will lie somewhere on the context.

How do I actually draw something?

We'll start with a very simple drawing operation. Our task is to draw two filled rectangles, covering opposite quarters of the Component's face. Like this:



When we were laying out the widgets earlier, we gave exact numbers, based on the size of the Component. Here, we're going to imagine we don't know the size yet (and that's true - remember that this Component is actually a little bit smaller than the DialogWindow).

We will be using the following function to draw these rectangles:

```
Graphics::fillRect (x,y, width,height);
```

This is identical to the Component::setBounds() function we used earlier, where we gave a top-left corner and a size. Add the following lines to your paint() function body:

```
g.fillRect (x1,y1, w1,h1);
g.fillRect (x2,y2, w2,h2);
```

The 'g' is the Graphics parameter we've been given to play with. Also, you'll notice that this is the first time we've used a '.' to call a function. This isn't a big deal, just remember that in this case we're using a reference, whereas so far we've just been using pointers. Now it is time for a task!

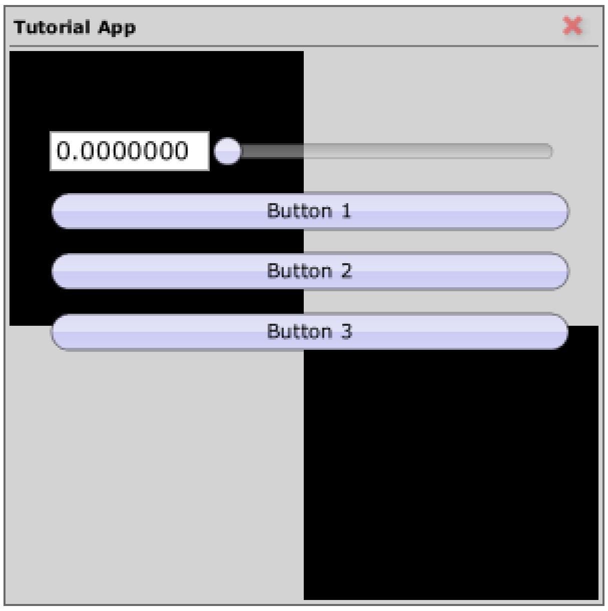
Task

Your mission is to calculate the x, y, width and height values for both rectangles! Work them out, and put them in place of the corresponding parameters in the function calls you just wrote.

Here are some tips:

- You can use 'getWidth()' and 'getHeight()' as numbers.
- Each rectangle is half the width and half the height of the context.
- One starts at the top-left corner, the other starts at the centre.
- Both rectangles are the same size.
- You can use variables to store values.

Remember, we're not using exact numbers, just proportions of the context's size. That means we'll have formulae using the context dimensions. Feel free to use a pen and paper to work them out, even though they are really simple! It's a good habit to have! Once you've got them worked out, put them in and compile the program. Run it, and take a look at the result (The solution is in the answers appendix).



There are several important things to notice about the result. One is that the rectangles are both black. The other is that our widgets appear on top of them!

Why am I painting 'under' the widgets?

In the paint() function, we are given a context that represents the Component *before* any child Components have been added. All of our drawing operations will be carried out, then the children will be drawn on top. What if you wanted to paint *on top of* the child Components? You *could* get around it by drawing on a child Component, and having that be the last drawn child... but that would be remarkably silly! The solution is simple, using an identical function called 'paintOverChildren()'. This has the same form and capability as the paint() function, except that it is drawn *after* all the children. We're not concerned about that though, and you're unlikely to use it often.

Why is everything black?

The rectangles we've drawn are both black, and drawn on top of grey – the colour of the DialogWindow. You may remember that a Component starts out completely clear, the grey we see is what the Component behind looks like. We've not told the rectangles to use any colour in particular, and black is as good a default as any! We can of course choose any colour we like, and this is made easy with the Colour class – and even easier with the Colours class.

A 'Colour' object represents a colour. A colour is identified by three light values (amounts of red, green and blue) and a transparency (or 'alpha') value. The Colour class gives us a bunch of handy functions to get Colours with slightly different values, which we shall explore shortly.

For now, we'll skip 'creating' our own Colour objects (because we don't need to do that just yet!), and instead make use of the 'Colours' class. 'Colours' is basically a palette of predefined colours that we can get pick from by name.

Examples of Colours...

```
Colours::lightblue;
Colours::red;
Colours::seashell;
```

These are all listed in the documentation.

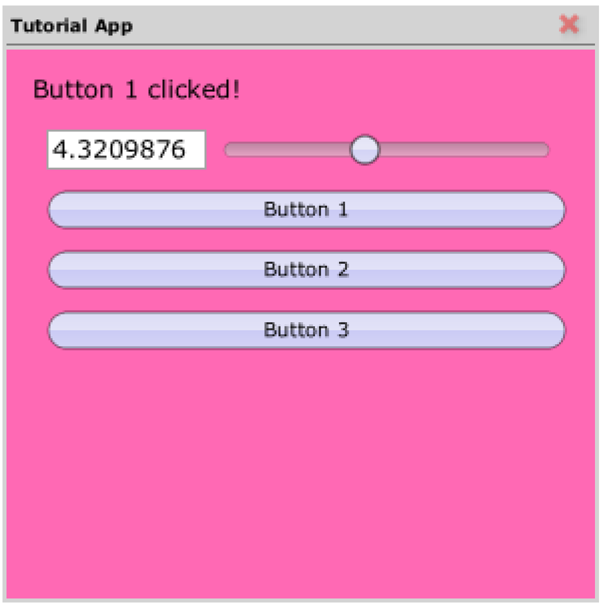
We can set the colour of our painting operations simply by calling 'Graphics::setColour()' beforehand. Add the following lines after the fillRect() calls:

```
g.setColour (Colours::hotpink);
g.fillRect (0,0, getWidth(), getHeight());
```

Notice how the Colour is retrieved from Colours 'on the spot', using the scope resolution operator '::'. This will draw a big pink rectangle over the whole context. Note that this can be achieved in one line using this:

```
g.fillAll (Colours::hotpink);
```

Try compiling and running both versions. You'll notice that they both look the same, like the image below:



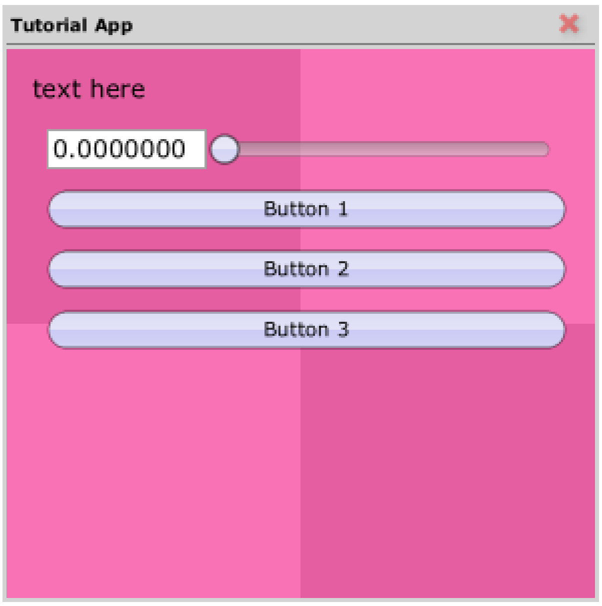
Very pretty indeed! You notice of course that the black rectangles have gone; this illustrates that everything is drawn *in order.* If you draw something you'll draw on top of the previous operation. If you used a transparent Colour, you'd still see them.

The Colour class, as mentioned previously, has a number of functions which can be used to alter the colour it represents. So far we've just taken a nice pink from the presets. Have a look at the documentation – you can make it brighter, darker, more transparent, etc. Note, though, that these functions don't actually change the object! They return a modified Colour, but the object is the same (unless you specifically reassigned it).

So, use one of these functions to try drawing our pink rectangle with a transparent pink Colour. This requires setting the *alpha* value to a proportion (between 0 (clear) and 1 (opaque)). Alter the line where you set the Colour, so that your Colour setting looks like this:

```
Colours::hotpink.withAlpha (0.9f);
```

The 'f', in case you didn't know, goes after a decimal value to indicate that it is a float constant. Build it, and look at the new panel.

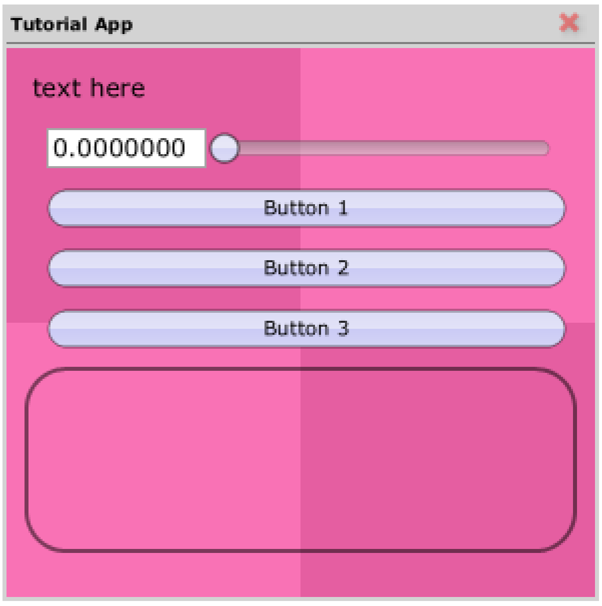


We can see how the Colour functions can be used to modify a Colour, and we can also see proof that the stuff 'behind' is still being drawn.

We'll now draw a little 'border' rectangle in the bottom section. We'll draw a 'rounded rectangle', with soft corners. Also, not that this is 'draw' and not 'fill' – this means that we'll just have an outline. We'll use a slightly transparent black, so that it blends with the background:

```
// set a transparent black colour...
g.setColour (Colours::black.withAlpha (0.5f));

// draw a rounded rectangle (2px thick, 20 'round')
g.drawRoundedRectangle (
 10, 160, getWidth()-20, getHeight()/3, 20, 2);
```

You should have the following:



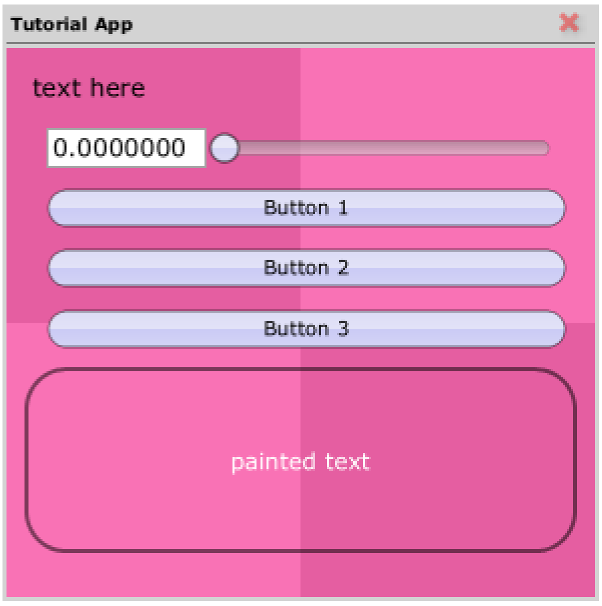
How can I draw text?

Lastly, draw some text inside this border. The absolute easiest way to do this is to use the 'drawFittedText()' function. This takes some text and draws it inside a rectangular region (specified by top-left and size, as usual). You also specify the justification (e.g. centred, left, etc.) using a simple Juce enumeration. The other thing you give it is a maximum number of lines for the text to take up.

Add these lines after the rounded rectangle has been drawn:

```
g.setColour (Colours::white);
g.drawFittedText (
  T("painted text"),
  10, 160, getWidth()-20, getHeight()/3,
  Justification::centred, 1 );
```

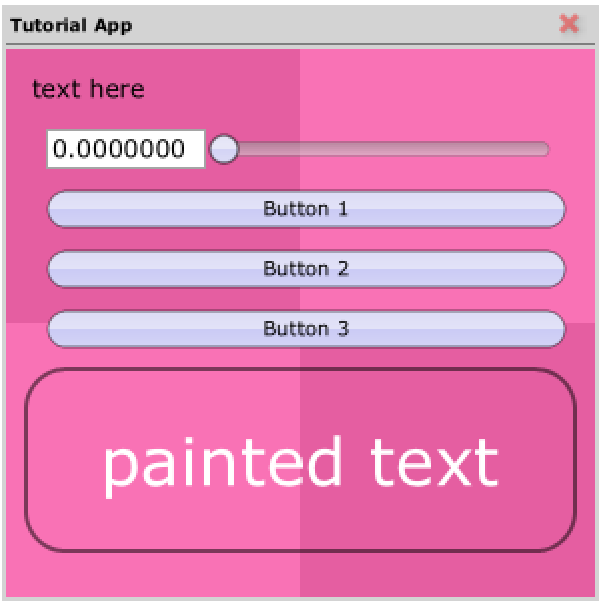
Notice that it's given the same rectangle bounds as the rounded rectangle we just drew. Compiling and running this gives us:



So now we can draw text! But it's really small, compared to the space it's in. How can we make the text bigger? Well, when you're using a word processor, to change the size of your text you edit the font properties. There is a whole Juce class dedicated to fonts (called 'Font', interestingly enough!) but we're not going to look at it in detail here. The Graphics class has a shortcut function, which allows us to simply specify a new font height. Add the following line before the drawFittedText call:

```
g.setFont (40);
```

Again, compile it and see how it looks!



There are a lot of drawing operations available from the Graphics class, but they're irrelevant at this point. Feel free to play around, of course; try drawing some ellipses or lines.

One last thing I've not yet mentioned is that all the coordinate parameters are floating point values. Why would this be, if they're referring to pixels? Well, Juce draws with sub-pixel accuracy, and it's all anti-aliased. What that means, to those who aren't robots, is that everything can be smooth, sharp and precise. You don't have to restrict your coordinates to heavy 'quantised' blocks!

So, we've seen how Components can be drawn on, using the Graphics class. We've also seen that there are several other classes that a 'Graphics' can make use of (including the Colour and the Font). Perhaps by now you can imagine (and appreciate) the kind of operations the TextButton uses in its own paint() function. Don't worry if it seems beyond you – it won't be too long before you're easily making your own widgets, and in a later chapter we will do just that!

Now, however, it is about time we took a look under the bonnet, and learned how to use Juce's core classes...

Chapter 5 – Juce Strings

Now we know how to show pretty much what we need on the screen, we are in a position to experiment with the core Juce classes. By 'core', I'm referring to the bits that make your application *work*. Even without a GUI, most programs can still perform their tasks because there is code performing operations 'inside' the machine.

The first core class we shall investigate is the String. We've already had some encounters with them, but we've never actually held a String object and done something to it.

What is a String?

A Juce String holds text. If you're familiar with programming languages, you'll already know a bit about strings. If not, you may be surprised to learn of the chore that string operations used to be!

In 'the olden days' of C, string handling was a really cumbersome business. The C language can handle 'null terminated strings', which are arrays of any number of characters, where the whole lot is ended with a null (zero). If you wanted to make a 'string', you'd have to allocate enough characters (plus one for the null) to hold your text. If you wanted to join strings together, you'd have to allocate MORE memory (so that enough characters run in a consecutive sequence in memory), copy the needed bytes over, then free the old memory. Pretty complicated!

C++ comes with the STL (which is an 'official' set of classes), which includes a string class for making such things simpler. You may have already used it in your past C++ experience. Juce, however, effectively renders the STL obsolete for our needs (being a set of ready made utility classes), and comes with its own String class. And very handy it is too!

With the String class, you don't have to worry at all about how the memory is organised, or what might be involved in manipulating strings of characters. The interface is entirely logical and intuitive, which I hope to demonstrate in this chapter. You can really use the String as if it were a built-in C++ type, like an *int* or a *float*. You can create one anywhere and just use it however you need. Let's try it out. We'll give our MainComponent a String member:

In the private declaration area...

```
String text;
```

This enhances our MainComponent, giving it a dedicated space to store text. Now, in any of our functions, we can access this String just by using this '**text**' member object. This object will be created as the constructor is called. By default a String is created empty, but if we wanted it to start out holding a particular phrase or value, we'd just assign it here, like this:

```
String text = T("Text goes here!");

// or using constructor...
String text (T("Text goes here!"));

// or initialising after instantiation...
String text;
text = T("Text goes here!"));
```

Notice that it's perfectly valid to assign a value to the text *after* the variable has been created (If you've never made null-terminated-strings before, that won't be interesting to you!). We're going to allow our '**text**' to start empty.

Where can I use the String?

As just explained, you can assign the String whenever you like. Our String '**text**' is always available to use in any of our MainComponent member functions.

If we want to set it to a particular value, we can, assigning it by name:

```
text = T("This new text!");
```

That means we can do stuff with our String in the functions we already have. Currently our application uses the buttons to directly set the text in the Label Component. Let's adapt it to instead change the value of our String.

The function that we need to change is of course the buttonClicked() function. For this example, we're going to make the following changes to **button1** and **button2** only:

In buttonClicked() ...

```
if (button == button1)
{
   text = T("Monkey");
}
else if (button == button2)
{
   text = T("Tennis");
}
```

This will cause **button1** and **button2** to set the text. Right now though, this won't appear to do anything (the Label is no longer connected to our buttons). We need to make use of this String so we can see the result.

We're not going to use the Label, we're going to use our 'home-made' text display code (in the paint() function). The 'drawFittedText()' call currently uses a string literal.

Let's change that, so that we're instead drawing whatever text we have stored at the time.

In MainComponent::paint()...

```
g.drawFittedText (
  text,
  10, 160, getWidth()-20, getHeight()/3,
  Justification::centred, 1 );
```

Compile and run this program. You should probably be puzzled, because the buttons don't do anything. The buttons *are* setting the String, let me assure you of that! This is time to become aware of an important fact when building GUIs. Changing the value of a variable does not cause the interface to update. You can logically reason this out for yourself (your code could change the values at any time, and you don't want the interface to constantly be updating itself when you don't want it to). In our buttonClicked() function, we're setting the text, but we also need to update the GUI. This is called "triggering a repaint", and will result in the new value of the String being displayed.

We want it to update whenever we click any of the buttons. Though our function has separate code blocks for each button, there is also space for more code *after* the checks; it'd be handy if the Component were to always redraw itself whenever it receives a button message. Add this:

At the end of buttonClicked()...

```
void buttonClicked (Button* button)
{
   if (button == button1)
   { ... }
   else if (button == button2)
   { ... }
   else if (button == button3)
   { ... }

   repaint ();
}
```

Compile this, and you should see that the top two buttons will now change the text on the bottom. The third button still changes the Label. In case it wasn't entirely clear, what we've done is call 'repaint()' (the Component's function for getting it to update) every time buttonClicked() is called, after the buttons have had their individual effect.

So we've seen how we can *assign* a String, and we've seen that we can paint its value on our Component's face. It's now time to see how the String can be really used.

What can I do to a String?

Before we do something with the String, we're going to change the behaviour of our last TextButton. We're going to make it act like a 'toggle' button, meaning that it will stay in when we click it, until we click it again (and then it will be 'off'). The Button class it derives from makes this very easy.

The function we want to use is:

```
button3->setClickingTogglesState (true);
```

This is a configuration we want this button to take when it is created, so we put it in the constructor. Put the above line in the MainComponent constructor anywhere after that object has been created (the end will do fine). Compile it, and notice the behaviour when you click on that button.

We can now toggle between some kind of 'mode'. We can say that, if the button is 'on', we do one thing; if the button is 'off', we do something else. To find out **button3**'s 'toggle state' (whether it is on or off), we use this:

```
button3->getToggleState ();
```

Is that simple or what? That function returns a boolean (true or false) value. If it is true, it means the button is 'on'. Remember that a function call effectively gets 'replaced' by the value it returns. That means we can use this call anywhere we would do a conditional check (or we could assign a bool variable with the result of the call).

We're going to use the state of this button to choose whether the text is shown in upper or lower case. One thing we could do is set the value of the String in the button call. This would be like the assignments for **button1** and **button2**, but would have a conditional check on the button's state to decide what to do. We don't want to do that here though. We've already seen that we can assign a String, we want to try something new!

Before we code the behaviour, let's change the text on the button to reflect its new purpose. In the constructor, where **button3** is created, change the text to say "display mode" or something to that effect.

Now, instead of changing the buttonClicked() behaviour for **button3**, we're going to alter the paint() function. We can check the state of the button at any time, not just when it has been clicked. Find the following line:

```
g.drawFittedText (
  text,
  10, 160, getWidth()-20, getHeight()/3,
  Justification::centred, 1 );
```

This draws the exact contents of **text** to the panel. We don't want to change '**text**' itself though, we just want to change how the text is displayed. For this, we're going to use a temporary (local) String variable.

Just before the above function call, we'll create a String object.

```
String textToDraw;
```

Now, every time paint() is called, a temporary String object will exist for us to use. As you may suspect, it's probably not the best idea to do too much memory allocation in 'paint()' for day-to-day programming, as you don't want your repaints to be a big CPU drain! A few variables shouldn't be a problem though.

How can I modify a String?

We're going to give this object a modified form of the text we have stored, which calls for some String functions.

To get an upper or lower case version of a String, we can just use the following function calls:

```
text.toUpperCase ();
text.toLowerCase ();
```

In the last chapter, I pointed out that, in the Colour class, functions like these don't actually convert the object you're calling it from – they just return a modified temporary which you can use for assignment. This is exactly the same here with the String class. If you wanted to actually *convert* the String to upper case, you'd say:

```
text = text.toUpperCase ();
```

We're not going to convert it though. We're going to assign this converted version to our **textToDraw** local String. We're going to choose the conversion by the state of **button3**. That means the following code:

In paint(), before the text is drawn...

```
if (button3->getToggleState ())
{
   // button is ON... (getToggleState() == true)
   textToDraw = text.toUpperCase ();
}
else
{
   // button is OFF...
   textToDraw = text.toLowerCase ();
}
```

We have now got a local String which exists for the duration of the function. Its value is set (according to the state of **button3**), and then it is painted. Except that it isn't yet! We're still drawing the main **text** String! Change the String parameter in drawFittedText() from **text** to **textToDraw**. Compile and run it.

Now **button1** and **button2** set the text, and **button3** just changes the case of the displayed String. Remember too, that the String member **text** is not actually changed by **button3**.

Many of the other String functions can be used in this way. In the documentation, any String functions that return a *const String* will return a modified version of the String, leaving the original intact. Also, the fact that they return String objects means we can stack these functions up. For example, if we wanted to have a capitalised, quoted version of a String, we could do this:

```
textToDraw = text.toUpperCase().quoted();
```

A lot of these functions are quite specialised but a quick read of the documentation should explain how they work. There's little need to cover them all here!

How can I combine Strings?

So, these functions provide one way for us to alter Strings. Another thing that you'll find yourself doing a lot is *combining* Strings. The String class has 'overloaded operators', which means it has special functions that are called when you use symbols like '+' or '='. Because of this, it's really easy to join Strings – you literally add them together!

Let's demonstrate this by 'doubling' the text when **button3** is pressed.

In paint()...

```
if (button3->getToggleState ())
{
   // button is ON... (getToggleState() == true)
   textToDraw = text + T(" and ") + text;
}
else
{
   // button is OFF...
   textToDraw = text;
}
```

This shows two things; you can add several items together at once (as many as you like!) and you can also mix string literals and String objects.

It's important to bear in mind exactly how the T("") macro works when using them in String assignments. While we've so far found that we can treat a T("text") string literal as if it were a Juce String (when calling functions, for example), it's actually not a String. The reason it matters here is because you can't 'add' two of these together like you can with Strings. What a T("") actually becomes in your code is a pointer to a null terminated string. Adding this to a String is fine because it knows how to deal with them, but you can't add two together, because they won't understand what want them to do. In any case, there's little reason to want to put two together (when you can just combine their contents into one), but you should be aware of it.

Just to make the point clear, you can do this:

```
String text = T("Name: ") + name + T(" -rank A");
```

But you can't do this:

```
String text = T("Name: ") + T("haydxn") + T(" -rank A");
```

You can, of course, change the T("") for a String("") if you absolutely need to do something like this, but you should probably see how it's not going to ever be an actual issue.

How can I use a number as a String?

Remember when we tried to display the value of the Slider on our Label? The Label Component will only draw a String value, but we had a number. The way we solved it was to use the String constructor like a cast.
If you wanted to write a sentence involving some numerical variable, you would do something like this:

```
int value = 10;
String text = T("The value is ") + String (value);
```

The String takes care of converting the number into text, so that it can be manipulated as a series of characters rather than a number.

An integer value is straightforward – the number '9' in text form is just the character '9'. The two characters '4' and '2' (in the correct order) are all that is ever needed to represent the number '42'. With floating point (fractional decimal) numbers, there is not always one way of displaying the numbers – think of pi as an example! How does the String class know how many decimal places to use when we try to use a float (or a double) as a String? Well, you just tell it!

We've already turned a double into a String, in the changeListenerCallback() function. This constructor actually allows us to supply a second parameter (check the documentation), so make the following alteration.

```
void changeListenerCallback (void* changeSource)
{
   if (changeSource == slider)
   {
      double value = slider->getValue ();
      label->setText (String (value, 2), false);
   }
}
```

Now, when you adjust the Slider, the label is updated with a rounded version of the value (to two decimal places). Easy!

So we can use any number as a String. What if we want to use this String as a number?

How can i use a String as a number?

If the user has typed something in, it is going to be stored initially as a String (because the user entered a string of characters). If we wanted to use the number they've entered in any sums, we'll need to convert the String first. There are a few simple functions for this purpose, and they can all be used in the following manner:

```
int value1 = text.getIntValue ();
int64 value2 = text.getLargeIntValue ();
float value3 = text.getFloatValue ();
double value4 = text.getDoubleValue ();
```

Now, before we test these, we're going to do a little tinkering with our app. So far, we've only got TextButtons and a Slider for user input. What I've kept secret, however, is that the Label is a handy user input widget in its own right!

Task

The docs show that a Label can become a TextEditor when it is clicked. First, however, it needs to be configured. This is what you must achieve to complete the task.

Here are some clues:

- You must add two Label function calls to the constructor.
- We want to edit the Label by clicking on it once.
- We saw earlier that the Label is a ChangeBroadcaster.
- We'll want to know when the user has entered text.

The two lines you'll need are in the answers appendix. Have a look through the Label class documentation and see if you can figure them out for yourself.

Once you've converted the Label, try building the app. If you click on the Label, you'll now be able to type something in! Who'd have thought we'd have such features built in to our quiet, unassuming test app!

We shall demonstrate how the String can be used as a number. Hopefully in an earlier chapter, you tried to set the value of the slider using the setValue() function. Don't worry if you didn't, you're about to!

The Label is a ChangeBroadcaster. You should already have reached the conclusion that a change message will be sent if the user types something in (if you haven't – *that's what happens!*). Our reactionary code then will live in the changeListenerCallback() function:

```
void changeListenerCallback (void* changeSource)
{
   if (changeSource == slider)
   {
      // the slider has been adjusted...
      double value = slider->getValue ();
      label->setText (String (value), false);
   }
   else if (changeSource == label)
   {
      // text has been entered...
   }
}
```

This gives us somewhere to respond to the change. What do we want to do when text has been entered? Well, we're going to try to take a numerical value from the String, and then use that to set the value of the slider. First of all, we need to get the actual text from the Label:

```
label->getText ();
```

This returns a String of whatever text the Label is currently displaying. Remember that the function call above, as it returns a String object, can be treated as if it IS a String object. Thus, if we wanted to get a double precision floating-point value from the text, we can say:

```
double value = label->getText ().getDoubleValue ();
```

Notice again how we can 'stack up' these function calls. We're getting the double value of the String retrieved from getText(), and we're assigning it to a variable named '**value**'.

We'll also want to set the value of the slider, using the setValue() function. Here are the amendments you need to make:

```
void changeListenerCallback (void* changeSource)
{
   if (changeSource == slider)
   {
      // the slider has been adjusted...
      double value = slider->getValue ();
      label->setText (String (value), false);
   }
   else if (changeSource == label)
   {
      // text has been entered...
      double value = label->getText().getDoubleValue();
      slider->setValue (value, false);
   }
}
```

Note at this point the importance of the 'false' parameter in setValue(). As pointed out earlier, this determines whether or not this assignment will trigger a new message being called. If it were true, then we'd cause the Slider to call this function again, whereby its value would be pasted into the Label! This kind of circular behaviour is rarely what you want, so you'll want to keep an eye on those types of parameter.

Now if you compile and run this application, try entering a value. It will happily accept decimal values. If you enter a value outside the range of the slider, it will be clipped. If you enter text that is *not a number*, then you get zero. That's handy – built-in error checking!

We're almost done with the String class now. Before we move on to something more interesting, I'll quickly point out two things that the String class can be which you'll undoubtedly make use of at some stage.

How do I know how long a String is?

To get the length of a String, you'd say something like this:

```
int x = text.length ();
```

How can I clear a String?

If you want to clear a String, or check whether a String is empty, you can use *String::empty*. For example:

```
if (text == String::empty)
{
   // there is no text...
}
else text = String::empty // clear the text!
```

The rest of the String functions are fairly self-explanatory, but if you get stuck with the docs, just ask a question on the forum. For now, we know enough of the String basics to be able to use them all around our programs. Let's just recap what we've learned about Strings:

- A String holds any amount of text.
- We can join Strings together with a '+'.
- We can assign Strings with an '='.
- Strings can return modified versions of themselves.
- Strings can be converted to and from numbers.
- We can easily find out how long a String is.

Now that we've got an understanding of the String class (a fundamental piece of Juce), we are in a good position to investigate some more classes. Because of the structure of this tutorial, I have decided that your next class subject will be the File.

In the next chapter, we'll learn how to save and load stuff! This may seem like an odd choice of topic, but the order will make sense soon enough!

Chapter 6 – File handling (part 1)

Now that we have the tools to get input from the user, it's time to look at how we might go about storing or retrieving that data. This means that we are going to learn about Juce's File class.

This chapter also has a basic 'project' application. By the end of this chapter, you should have a working 'note pad'.

What is a File?

The Juce File class allows you to hold references to files on your computer. When you create a File object, you create a link to a single item on a drive. The item may exist, or it may not, and it may be either a file or a folder.

That's the basic principle behind them. We're going to begin our study of the class by learning to open a file that already exists. First, load up notepad (or something similar) and create a plain text document. Type in some arbitrary nonsense, and save it in a simple location with a memorable filename (for instance, "C:\filetest.txt").

Before we actually try to use this file, we'll do some quick spring cleaning. The MainComponent we've been working on is full of things we don't need, so we'll empty it out and start fresh. For this chapter, we'll build upon our first "Starting Point" code (which has no child Components yet on the interface).

The last chapter introduced a text entry widget, via the Label. The Label is not the only text entry Component though. In fact, it's actually not really one all by itself. The reason you can use it like that is because it can turn itself into a TextEditor – another Juce Component type. The TextEditor has all the text entry features you could need!

Let's add one to our MainComponent.

In the private declaration:

```
TextEditor* text;
```

And in the constructor:

```
text = new TextEditor (T("Editor"));
addAndMakeVisible (text);
```

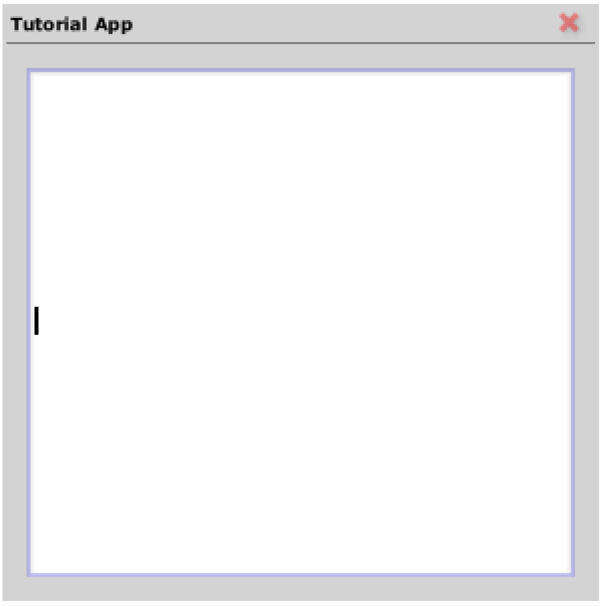
Make sure you also have the 'deleteAllChildren()' call in the destructor, so that our child Components are destroyed.

We of course need to position it on the panel too, so we must add the resized() function and set the editor's bounds:

In the public section:

```
void resized ()
{
   text->setBounds (10,10,getWidth()-20,getHeight()-20);
}
```

Note that we're giving it a 10 pixel margin, making use of the size of MainComponent. Compile and run this.



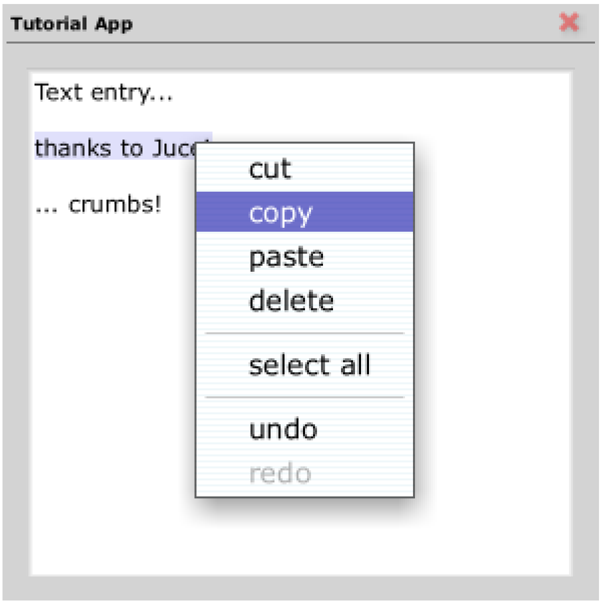
You should notice a couple of odd features of this text entry window we've created. Firsly, the flashing caret is halfway down the box. Secondly, it only types one line, and you can't press 'enter' to create a new one. As far as notepads go, this is poor!

But don't worry, this is just the default TextEditor behaviour. By default, it expects to be used as a single strip of text (as we saw when we used the Label). We can change that though, with a few simple function calls just after we've created it:

In the constructor:

```
text->setMultiLine (true);
text->setReturnKeyStartsNewLine (true);
```

These two lines of code are entirely self-explanatory! Compile again, and try entering some text.



Notice also that this editor has a right-click menu built in! Without even doing anything, we've got copy/paste, and even undo/redo features.

How do I open a File?

We now have a big pad to display and edit text on. What we're going to do next is learn how to read the file you created earlier, and display it on our editor.

We need to create a File object to 'be' our file. We can create a File object anywhere, just like we can with a String. Just like a String is created empty automatically, a plain File object doesn't actually refer to anything (its file target is non-existent).

We need to assign it to our file, which we shall achieve using a String of the file's full path. For example, either of the following methods work:

Creating the object then assigning it:

```
File myFile;
myFile = T("C:\\filetest.txt");
```

Assigning via the constructor:

```
// or...
File myFile (T("C:\\filetest.txt"));
```

Notice that the '\\' is not a typo! In C, null terminated strings use 'backslash constants' to indicate characters that you can't type on the keyboard. For instance, a 'newline' is '\n'. The '\' means *"the next character is a special code"*, which also means that you can't type a '\' normally! To actually use a '\', you put two together (to show that it's not a code after all, and you actually want a slash, as it were).

We're going to go for the former of the above two examples (assigning *after* creation), because we're going to keep our File as a member of MainComponent. Thus, we'll always have a place to remember a particular file. Add the member object in the private section, and assign it in the constructor:

In the private declaration:

```
File currentFile;
```

In the constructor:

```
currentFile = T("C:\\filetest.txt");
```

Our program, when it runs, will now have a link to this file. It's not actually tried to open the file or anything yet though, it just knows where it is.

To open it for our needs is incredibly simple. Laughably so. Before I get onto that though, I shall point out that it's important to check your files! As I mentioned earlier, a File object may be linked to a file that doesn't even exist. This may be the case here (for example, if you typed in the filename incorrectly), so we'll get into the habit of checking it first.

If a file doesn't exist, then comparing a File object with **File::nonexistent** will return **true**. This is one way of checking, but there are several. The one we're going to use is a simple bool returning function. It is used in the following manner:

```
If (currentFile.existsAsFile ())
{
   ...
}
```

Now we can perform a task only if the file definitely exists! We want to open the file in our editor, and for that, we'll create a function of our own (with an appropriate name).

Create the following function.

```
void openFileInEditor ()
{
}
```

What is it that we need to do in this function? Well, we've seen that we should have some kind of safety check in there (make sure the file exists). Then, we want to read the text from the file, and put it onto the editor.

The editor part of the task is easy. The TextEditor has the same 'setText()' function as Label, so we can use that. We just need to get a String of the text from the file. How can we do that?

How do I read text from a File?

This couldn't be easier. Look through the File documentation, and see if you can spot the function for yourself. It's just about as obvious as you can get. In case you're having a hard time finding it, here's the answer: you can get a String (containing the whole file's text) just by making the following call:

```
currentFile.loadFileAsString ();
```

Did I say it was simple? That function returns a String, so we can use that directly in our setText() call. Give your function the following form:

```
void openFileInEditor ()
{
   if (currentFile.existsAsFile ())
   {
      text->setText (currentFile.loadFileAsString());
   }
}
```

As this is a test, we want the file to be opened automatically when we launch the app, so put a call to this function in the constructor (after the File has been assigned):

In the constructor:

```
...
currentFile = T("C:\\filetest.txt");
openFileInEditor ();
```

Assuming that you've got the path typed correctly, when you compile and run the app, you will see the contents of your file in the TextEditor. Brilliant!

How do I choose a file?

In real life (the life of a programmer at least), you almost never know an exact file path within your code. For example, it's unlikely that anyone actually has a file called "filetest.txt" in the root of their C drive, so it would be silly to release a program that only opened that one file.

No, generally in a program there is a way for the user to choose a file. This is another thing that Juce makes easy for us! And it requires the introduction of another class, the FileChooser.

We'll make another function, called 'chooseFileToOpen()'. Let's think about what we need it to do for us. We want this function to pop up a window allowing the user to locate an existing file. We then want the chosen file to be stored in our File object so we can open it with the function we wrote earlier. Does that sound complicated? It shouldn't, but even if it does, don't worry! Our function won't return anything, or take any parameters, so it'll look a bit like this:

```
void chooseFileToOpen ()
{
    ...
}
```

Now, what we need to do is very straightforward. If we need to bring up a file selection window, we need to create a FileChooser object. If we just make a local (function scope) instance, we know that it will be tidied away after the function has finished, which is what we want. The documentation for the FileChooser class perfectly explains how it should be used. Basically, you call one of the 'browse...' functions from the FileChooser object, and respond based on the result. If the browse function returns false, it means that no file was chosen, so we can put this call inside an 'if' statement to quickly obtain the result. The actual File chosen (if one is chosen) is retrieved with a separate function call.

Here's the code we need to bring up a FileChooser (for opening a file) and open the result:

```
void chooseFileToOpen ()
{
   FileChooser chooser (T("Choose file to open"));
   if (chooser.browseForFileToOpen ())
   {
      currentFile = chooser.getResult ();
      openFileInEditor ();
   }
}
```

Now we just need to change the constructor a little to make use of our new feature. That means stripping out the line where you tell it the old filename (yes, just delete that line), and changing the function call from 'openFileInEditor()' to 'chooseFileToOpen()'.

Run this program, and marvel at how you can now suddenly choose the file yourself! Notice in particular that you can now trigger this with just a single function call. From this thought, move on to the obvious idea that this is something you'll want to be able to do at will (not just when the program starts). By now you should know exactly what it might take to have an 'Open file' button. Time for another simple task then!

Add a button to MainComponent that will allow the user to open a different file. Chapters 2 and 3 have all the information you'll need (you've done this before!).

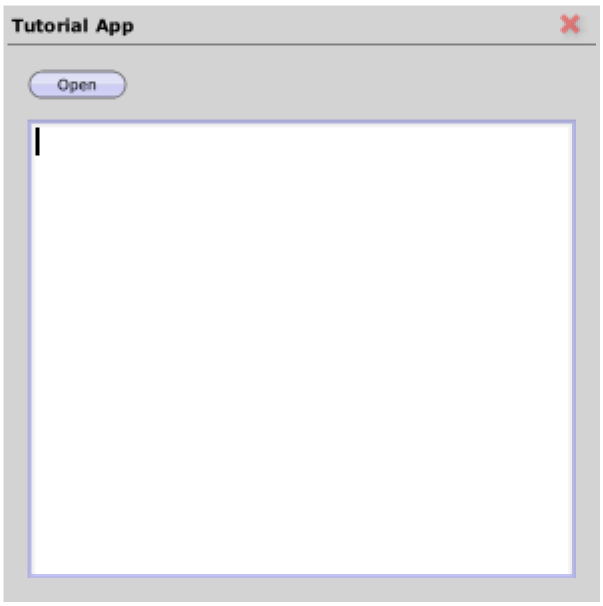
Here are some clues:

- You must create a pointer for the TextButton.
- You must create and add the TextButton.
- You need to be able to listen to Button messages.
- You need to position the TextButton.
- You'll need to reposition the TextEditor.

- And of course you'll need to respond to the button...

See if you can do this simple task. Example answers are in the appendix section.

If you've done it properly, you should hopefully end up with something a bit like this:



The actual layout of the Components isn't particularly important at the moment, so don't worry if yours looks different.

Now, we have a program that allows us to load any file we please as text! Perhaps you forgot to take out the call in the constructor (the one we did before the task) and had it ask you for a file when you load the program. If you did, great! It's always good to learn things from experience. So, our program loads up (empty) and presents us with a button to load existing text.

We'll take it to the next logical step, and add a button for saving. We don't know how to save yet, but we'd do well to have a button ready to try it out with for when we do!

Next task! Add another button, in the same way as before. Make sure you have a relevant section within buttonClicked() for it!

How do I save text to a file?

We now want to be able to save the contents of our editor to a text file. In simple terms, that means taking a String (the editor text) and writing it to a file.

What we shall do is have a similar structure to the 'open' operations; one function will perform the operation, another will prompt the user to select a file to use.

For the time being, let's assume that we've already managed to have the correct file chosen. The File object **currentFile** has been set to the target file, so we know where we need to write to. Our 'saveFileFromEditor' function will use this target and write the text.

How will we write the text? Well, the File class has a function called 'appendText()', which will write a given String to the end of the current file. If the file exists already, of course, this would result in a merge of whatever data it held and the newly written data. We want to make sure that, when a String is written to our File, the file is empty. The easiest way to do this is to erase it and create a new one! If the file doesn't already exist, then we just need to first create it (remember that a File object is just a reference to a potential file).

So, we'll definitely need to create a file at some stage in our function! That will make use of the 'create()' function. Note from the docs that this function won't do anything if the file already exists. We also need to know how to delete a file, in case it does exist, so that we can create a fresh one in its place; this is done with the 'deleteFile()' function.

```
void saveFileFromEditor ()
{
   if (currentFile.existsAsFile ())
      currentFile.deleteFile ();

   currentFile.create ();
   currentFile.appendText (text->getText ());
}
```

That's our 'save' function. We now need a chooser function, just like the one for the open button. In fact, it's so much like it that we can get away with copying and pasting the original, and making a few slight edits. Below is the new one, with **bold** type highlighting the bits that have changed:

```
void chooseFileToSave ()
{
   FileChooser chooser (T("Choose file to save"));
   if (chooser.browseForFileToSave ())
   {
      currentFile = chooser.getResult ();
      saveFileFromEditor ();
   }
}
```

Now we just need to put a call to this function in buttonClicked():

```
void buttonClicked (Button* button)
{
   if (button == open) chooseFileToOpen ();
   else if (button == save) chooseFileToSave ();
}
```

Compile and run this, and you'll delight at the power of your application! You can save and load text files!

Notice anything peculiar about the save function? It's actually behaving more like a 'Save As...' feature, as it always asks you where you want to save. In most applications, 'Save' will save the currently open document to the file it belongs to, whereas 'Save As...' will prompt the user for a location. If a document is newly created, it will not yet have a file allocated; a call to the normal 'Save' will in this case require a prompt.

Try to improve your 'text pad' application in this respect. You should end up with four buttons (New, Open, Save, Save As...). Notice the difference between the "Save" and "Save As" operations, and the functions we've made. Also, try to see the importance of the 'New' command (think of the 'Save' action behaviour).

You will, of course, want to spend time making sure it looks the part too! Get your buttons lined up nicely. There is actually a more appropriate Component (the MenuBarComponent) but that's a bit beyond this tutorial for the time being. It should of course go without saying that any of the other classes can be learned through experimentation (that's how I learned them), so if you're feeling adventurous, try some of them out.

What else can I do with Files?

We've only covered file selection and text access so far. There is obviously a bit more to file handling than that though! The File class gives you functions that allow you to perform pretty much any OS file operation you need (you can copy, move, create, delete, run, etc...). As well as 'doing stuff' with (and to) files, you also have a selection of handy functions to help you navigate your way around the filing system. Your task is not always as simple as "ask the user for a file", as you'll no doubt see soon enough!

This stuff is very important, of course, but is difficult to demonstrate with our current application. We are now going to take a break from covering specific Juce classes, and build something 'old-school' that might make our lives a bit easier.

One last thing before we end this chapter. You'll no doubt find yourself growing annoyed that your 'text pad' app is a fixed size. It's time we made things resizable! Just go into the MainHeader.h file, and change the 'resizable' setting to **true**. A recompile should give you a resizable window. This will also indicate whether or not you've got your resized() function working correctly!

Chapter 7 – Intermission

In this chapter, we're not going to focus on any one particular Juce class. We are instead going to look at ways in which we can test the other internal systems.

In the first few chapters, we played around with a bunch of Components on our MainComponent. Each one could be used to do or show something, but only really one thing at a time. The label shows one piece of text, the slider holds one value, and the buttons each trigger one action. It was a nonsense program, but we learned how to do some key things, and how the Component system works.

The last chapter saw us building a basic application that actually had a purpose. In doing so, we saw practical uses for certain class functions (such as File::loadFileAsText()). Ideally, we'd be able to make a 'real' program to demonstrate all things, but that would take too long. There are a great number of functions that you can only appreciate the use of through practical examples, and you'll most likely need to use them at some point.

In 'the old days', we'd test out the workings of various functions and classes by using the *console*. Yes, that relic i mentioned in the introduction chapter. This provides an ever-present dumping ground for any old data you felt like generating. You don't need to add a Component to the screen just to have somewhere to try out a few processes. All you need to do if you have a console is perform some test, check the results and output some meaningful text so you can see for yourself what happened.

That was 'the old days'. In our modern age, we're using Juce, and it's making us very happy. How can we have the luxury of the console, without having to actually worry about any other libraries?

There are two things you can do to get this kind of functionality. The first is to use a debugger, and the second is to just make your own console. We're going to try both methods in this chapter.

How can I use the debugger as a console?

The day I discovered that I could use the debugger, my programming life got a million times easier. Sounds stupid, doesn't it! Fancy going for so long without ever trying it! Well, the fact that I did means that it's possible, so I'll presume here that you haven't really tried it (although I seriously expect that you already have long ago!).

I'm not talking about the memory checking tools a debugger gives you. I'm talking about the 'standard error stream' output. I use Microsoft Visual C++ (express edition) for my Juce development, and it provides a handy text output window that pops up when you compile and run a debug build. There is every chance that you don't have a debugger, and just code using a compiler and suitable text editor. If this is the case, then you'll not be able to test this part of the chapter, but I highly recommend looking into it.

Hopefully you understand about the difference between debug and release builds. You use one code base, but certain parts of it are absent in a release build; the debug build has extra checking code to help you identify what is actually going on behind the scenes. If you 'attach' your debugger to the program built in debug mode, you gain access to the internal state of the program, but also you hook into something called the 'standard error stream'. If your program is set to send text to this stream, it will show up on your debugger's output log (assuming that's how your debugger works!).

In Juce, we can very easily write text to this stream. We just need to use a simple macro (like the T("") macro). The macro is:

```
DBG (const String& text);
```

Note that it takes just a single String.

We can use this anywhere we like. For example:

```
for (int i=0; i<10; i++)
{
   DBG (T("The current number is ") + String(i));
}
```

This will send ten lines of text to the debug console, each ending with a number from 0-9.

The DBG() calls are not function calls, but macros representing function calls – this means that you can completely omit them from a release build. In case you don't understand how that is, read up on macros!

Basically, if it is in 'debug' mode, 'DBG()' will be replaced with an actual function call (to send a message to the debugger), but if it isn't (e.g. Release mode), it is replaced by a blank space (so nothing happens!). This means you can liberally plaster them all over your code, and not have to worry at all about using up too much CPU in release mode from doing "pointless" tests.

Try it out for yourself. Go into any of the code we've already done, and add some DBG() messages. You could, for example, do:

```
void foo ()
{
   DBG( T("Function foo() called!"));
   ...
}
```

Build a debug version, and look at what your debugger displays. If you have it set up correctly, it should be working right away. Note that I'm only familiar with the Microsoft debugger within Visual C++, so I can't offer any real debugger advice (this one just works without extra configuration, but I imagine some others may be more problematic).

How can I make my own console?

Yes, the alternative to the debugger (although it's not really practical when working on a *real* application – you really should get acquainted with a debugger for that!) is making your own console in Juce. Actually, there are probably a fair few alternatives, but we'll at least maintain a moderate belief that there aren't.

This next section is not just about building a Juce console, however. It's actually a cunningly disguised introduction to building your own Components, and making your own reusable base classes.

We've already made one Component (MainComponent), but we've not yet tried to make our own 'thing' to put inside it. We want to make a new 'thing', so we must first think about what that thing will be. The console is just one feature, so it doesn't take much time to decide what it needs; it just needs to display lines of text! We've already seen a Component that can display lines of text (the TextEditor), but it's not quite what we need. It's a brilliant start though! What if there were some way we could adapt it to act like a console?

How can I make a class that's like another class?

How much 'like' the TextEditor is our console? Well, it doesn't need to look any different, as it just needs to hold lines of text (we already know it can do this well). The only thing different is its behaviour. We just want to make something we can send lines of text to. We don't want it to be typed on (it's just for text output). That's about it, really.

Conveniently, we can take all the existing features of TextEditor and use them in our own 'TextConsole' class without having to butcher anything. We've already made something that did the same thing as something else (MainComponent has done the same things as Component, ButtonListener and ChangeListener), so perhaps you know what we're going to do!

We're going to derive TextConsole from TextEditor. The console will *be* a TextEditor, but it will also have its own unique properties.

How should I make my new Component?

One thing that we've so far done is put all our coding into one file. MainComponent.h has the whole class declaration and definition all together, which a lot of people seem to frown upon. Read up on header and implementation files if you're not already familiar with such matters. We'll mostly not bother with creating separate '.cpp' files, because our headers will generally be small enough to not warrant them. We will however, at the very least, keep 'unique' classes in separate header files.

Start up a new project, using the StartingPoint files. Create a new header file, called "**TextConsole.h**" (make sure it is in the same directory as the other source files).

This is a header file, so we should first 'protect' it from multiple inclusion with the 'header protection defines'. This takes the form:

```
#ifndef _TEXTCONSOLE_H_
#define _TEXTCONSOLE_H_

...

#endif//_TEXTCONSOLE_H_
```

This is something you can read up on, but basically it stops the file from being included more than once in any compile, using pre-compiler macros. Traditionally, you use the filename in uppercase with underscores; you want it to be unique to the file. Any text between the two blocks gets ignored if it's already been parsed. Put those lines in, and treat them as the beginning and end of the file. Any subsequent code we enter will go in place of the ellipsis (...).

Now we have a file ready for our new class. We are going to be using Juce code here, so we must #include it:

```
#include <juce.h>
```

Now we can use the TextEditor class in our declaration. We want TextConsole to *be* a TextEditor, so:

```
public:
   class TextConsole : public TextEditor
   {

   };
```

It always helps to put the brackets in there right away, primarily to make sure you don't forget the semi-colon at the end! Also, it's very important that you have this in the public section!

Looking through the TextEditor docs, we can see that there are no 'pure virtual' functions, and there are no required parameters in the constructor. That means our TextConsole is currently ready to be used – but only as a TextEditor. If there were pure virtual functions, we'd need to define them first. Also, if any constructors required parameters, we'd have to set them in our own constructor. Even though we don't have to, we'll give it a parameter anyway – just to show how.

We need to make a constructor for our class:

```
TextConsole ()
{
   ...
}
```

To give parameters to the TextEditor part of the constructor (remember it IS a TextEditor, so that constructor will be called too), we use the *initialiser list*.

```
TextConsole () : TextEditor (T("Console"))
{
    ...
}
```

We've given the TextEditor constructor some text, because that parameter is the Component name parameter. If you had more base classes that needed parameters, you'd separate each additional constructor call with a comma.

So we have our constructor, which currently has no code in its body. What do we need to do here? Well, think back to the last chapter when we first added a TextEditor. The first thing we saw was that we needed to reconfigure it using a few functions. These were called from the constructor of the Component it sat inside. If we wanted it to behave like that in *every* program we wrote, we'd have to put those lines in every time. Or would we?

The constructor is just a place where we tell it how it needs to start life. We could very easily put those functions from before into our constructor here; that would mean that it would exhibit that behaviour (multi-line, return creates new line) automatically! We don't want quite that behaviour though. What do we need from it? A quick think and a browse through the TextEditor class shows the following functions to be of use.

```
SetMultiLine (true);
setReadOnly (true);
```

Add these to the constructor. Because it is a TextEditor, we can just call these functions without an object (as they belong to us!).

Now if we were to create it, it would be completely useless to the user, but it would be able to display any text we told it to (using functions). You wouldn't be able to type anything on it, but you could call 'setText()' to change it from the code. 'setText()' is too inflexible for our needs though – we want to *add* new lines to it, not set the whole thing. We need to make a new function:

```
void addLine (const String& text)
{
   ...
}
```

This is as good a name for a function as we need. We'll want to use it to add a new line of text to our console. How will we achieve this? Another browse of the TextEditor class documentation is required!

There are two main functions we need. The most obvious is:

```
insertTextAtCursor (String textToInsert);
```

Which will put the text at the current cursor position. Our function will take a String parameter (**text**), so we'll be using that. However, we'll be wanting each bit of text to be a new line, so we must add a 'newline' character to the end. I've already explained how to make one of these, using a backslash code. We can simply say (**text + T("\n")**).

The other function we need is:

```
setCaretPosition (const int newIndex);
```

Which allows us to move the cursor; we'll always want to insert text at the end. We therefore need to be able to find the end. This is easily done – we just need to find the length of the String of text held in the editor. That means:

```
int size = getText().length();
```

Easy! So we just need to assemble these in our addLine() function:

```
void addLine (const String& text)
{
   setCaretPosition (getText ().length ());
   insertTextAtCursor (text);
}
```

Now we have a special TextEditor with 'an extra button' that can do something specific for us. Let's give it a try!

How can I use my new Component?

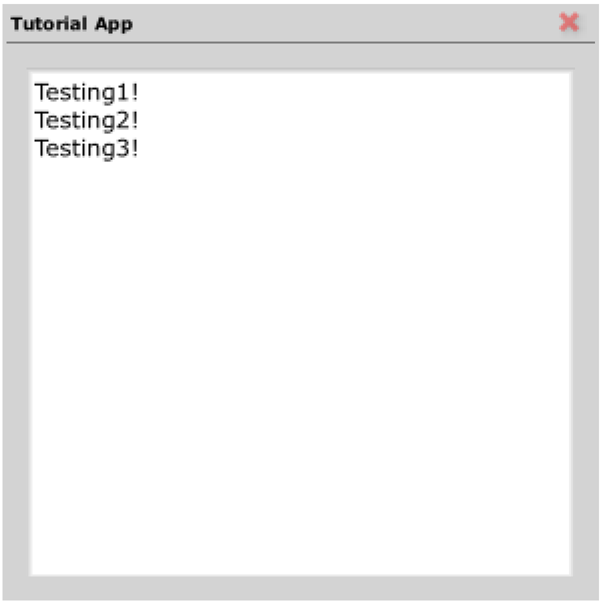
We've made this class in a header file, which means we need to include that file wherever we need it. In MainComponent:

```
#include "TextConsole.h"
```

Include the file before the declaration of MainComponent. That's all we need to do to be able to use our new toy. Now we just have to add it, in exactly the same way as we did with the TextEditor, and the TextButton, and the Slider, and the Label.

```
Class MainComponent : public Component
{
private:
   TextConsole* console;

public:
   MainComponent ()
   {
      console = new TextConsole;
      addAndMakeVisible (console);

      // a few test lines...
      console->addLine (T("Testing1!"));
      console->addLine (T("Testing2!"));
      console->addLine (T("Testing3!"));
   }

   ~MainComponent ()
   {
      deleteAllChildren ();
   }

   void resized ()
   {
      console->setBounds (10,10,
                        getWidth()-20,getHeight()-20);
    }

};
```

Compile and run this new application, and you should see your test lines sitting happily on the editor.



Testing1!
Testing2!
Testing3!

So we now have a working 'text output' console. It's not quite the same as a real console, or using the debugger. Some of you may be scowling at the page, because you know something I've not actually pointed out yet – Juce can do console apps just fine. Yes, that's right, you can make DOS style applications, but we're not here to learn that, are we? No, we're not.

I've not just wasted your time though. We've just learned how to adapt a Component into a new type to suit our needs, that we can now use anywhere. You can do this kind of 'additional' customisation with any existing class. All you need to do is derive from it, then configure and enhance to taste. We have, in truth, been doing something like this the whole time (with MainComponent), but doing it this way should make the Component model a little clearer.

The purpose of this TextConsole is to allow us to test core classes quickly and easily without having to build special displays for them first. I expect that you don't want to have to add new buttons, labels and things just to be able to try out some new class that I'm teaching you. Despite the TextConsole's usefulness in outputting information, we'll still need to add a selection of controls so we get a say in what's being tested.

How can I make my Component more useful?

Wouldn't it be good if we could make a 'testbed' Component, which had a console and several 'empty' controls. We could design it once (set up the buttons and things) and then just use 'new versions' of it (where we just design behaviour) for testing. This is what we are going to try now. In doing this, we shall see how we can make a more *general* class, with bits that we can change for our new versions.

Here is a reasonably simple task for you. Yes, I've stopped holding your hand for this one!

Task

Make a new file called **TestBed.h**, and put in appropriate protection instructions (#ifndef, #define and #endif), and include both the Juce header and the TextConsole header.

Create a new Component called TestBed. Give this Component a TextConsole, four buttons and a slider. Make sure the Component is capable of responding to all of the controls, but don't give it any action in particular to perform.

We will start with exactly the same stuff as the current MainComponent (which already has a TextConsole), so we can copy the code and edit what we need to. What we want is to have a Component called TestBed, which has a single TextConsole child.

Do this now (if you do use copy/paste, make sure you change MainComponent to TestBed in the new file!). Answer code is in the appendix (we of course take out the test lines, and we also want to change the bounds of the console so that we have space for buttons).

Now we have a Component that ... tbc

Answers to problems

Chapter 3

A) To set the value of the slider from a button click...

```
slider->setValue (53, false); // set value to 53
```

B) To set the text on another button from a click...

```
button1->setButtonText (T("different button 1!"));
```

Chapter 4

The simplest solution is as follows:

```
int w = getWidth()/2; // width of a rectangle
int h = getHeight()/2; // height of a rectangle
g.fillRect (0,0, w,h);
g.fillRect (w,h, w,h);
```

Chapter 5

The two lines to configure the Label as a user-input device are:

```
label->setEditable (true);
label->addChangeListener (this);
```

Chapter 6

Here are the steps that must be taken.

1 - MainComponent must derive from ButtonListener.

```
Class MainComponent : public Component,
                      public ButtonListener
{
   ...
```

2- MainComponent must be given a TextButton member pointer.

```
{
private:
   TextButton* open;
   ...
```

3- The TextButton must be created and added in the constructor, and MainComponent must register with it as a listener.

```
MainComponent ()
{
   ...
   open = new TextButton (T("Open"));
   addAndMakeVisible (open);
   open->addButtonListener (this);
   ...
}
```

4- The buttonClicked() function must be implemented.

```
void buttonClicked (Button* button)
{
   if (button == open)
   {
      chooseFileToOpen ();
   }
}
```

5- The button must be positioned on the panel.

```
void resized ()
{
   open->setBounds (10,10,50,15);
   text->setBounds (10,35,getWidth()-20,getHeight()-45);
}
```

[FURTHER SUGGESTIONS AND SOLUTIONS TO BE ADDED]

Chapter 7

The first code for the TestBed:

```
Class TestBed : public Component
{
private:
   TextConsole* console;

public:
   TestBed ()
   {
      console = new TextConsole;
      addAndMakeVisible (console);
   }

   ~TestBed ()
   {
      deleteAllChildren ();
   }

   void resized ()
   {
      console->setBounds (10,30,
                        getWidth()-20,getHeight()-40);
   }
};
```