Employing smart pointer classes

The `OwnedArray` class may be considered a manager of smart pointers, in the sense that it manages the lifetime of the object to which it points. JUCE includes a range of other smart pointer types to help solve a number of common issues when writing code using pointers. In particular, these help avoid mismanagement of memory and other resources.

Perhaps the simplest smart pointer is implemented by the `ScopedPointer` class. This manages a single pointer and deletes the object to which it points when no longer needed. This may happen in two ways:

- When the `ScopedPointer` object itself is destroyed
- When a new pointer is assigned to the `ScopedPointer` object

One use of the `ScopedPointer` class is as an alternative means of storing a `Component` objects (or one of its subclasses). In fact, adding subcomponents in the Introjucer applications graphical editor adds the components to the code as `ScopedPointer` objects in a similar way to the example that follows. Create a new Introjucer project named `Chapter03_05`. The following example achieves an identical result to the `Chapter02_02` project, but uses `ScopedPointer` objects to manage the components rather than statically allocating them. Change the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

#include "../JuceLibraryCode/JuceHeader.h"

class MainContentComponent :  public Component,
                              public Button::Listener,
                              public Slider::Listener
{
public:
  MainContentComponent();
  void resized();

  void buttonClicked (Button* button);
  void sliderValueChanged (Slider* slider);

private:
  ScopedPointer<Button> button1;
  ScopedPointer<Slider> slider1;
  ScopedPointer<Label> label1;
};

#endif
```

Notice that we use a `ScopedPointer<Button>` object rather than a `ScopedPointer<TextButton>` object for the same reasons we used an `OwnedArray<Button>` object in preference to an `OwnedArray<TextButton>` object previously. Change the `MainComponent.cpp` file as follows:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
  button1 = new TextButton ("Zero Slider");
  slider1 = new Slider (Slider::LinearHorizontal,
                        Slider::NoTextBox);
  label1 = new Label();
  slider1->setRange (0.0, 100.0);
  slider1->addListener (this);
  button1->addListener (this);
  slider1->setValue (100.0, sendNotification);

  addAndMakeVisible (button1);
  addAndMakeVisible (slider1);
  addAndMakeVisible (label1);
  setSize (200, 100);
}

void MainContentComponent::resized()
{
  button1->setBoundsRelative (0.05, 0.05, 0.90, 0.25);
  slider1->setBoundsRelative (0.05, 0.35, 0.90, 0.25);
  label1->setBoundsRelative (0.05, 0.65, 0.90, 0.25);
}

void MainContentComponent::buttonClicked (Button* button)
{
  if (button1 == button)
    slider1->setValue (0.0, sendNotification);
}

void MainContentComponent::sliderValueChanged (Slider* slider)
{
  if (slider1 == slider)
    label1->setText (String (slider1->getValue()),
                     sendNotification);
}
```

The main changes here are to use the `->` operator (which the `ScopedPointer` class overloads to return the pointer it contains) rather than the `.` operator. The components are all explicitly allocated use the `new` operator, but other than that, the code is almost identical to the `Chapter02_02` project.

Other useful memory management classes in JUCE are:

- `ReferenceCountedObjectPtr<ReferenceCountedObjectClass>`: This allows you to write classes such that instances can be passed around in a similar way to the `String` objects. The lifetime is managed by the object maintaining its own counter that counts the number of references that exists to the object in the code. This is particularly useful in multi-threaded applications and for producing graph or tree structures. The `ReferenceCountedObjectClass` template argument needs to inherit from the `ReferenceCountedObject` class.
- `MemoryBlock`: This manages a block of resizable memory and is the recommended method of managing raw memory (rather than using the standard `malloc()` and `free()` functions, for example).
- `HeapBlock<ElementType>`: Similar to the `MemoryBlock` class (in fact a `MemoryBlock` object contains a `HeapBlock<char>` object), but this is a smart pointer type and

supports the `->` operator. As it is a template class, it also points to an object or objects of a particular type.