Chapter 2. Building User Interfaces

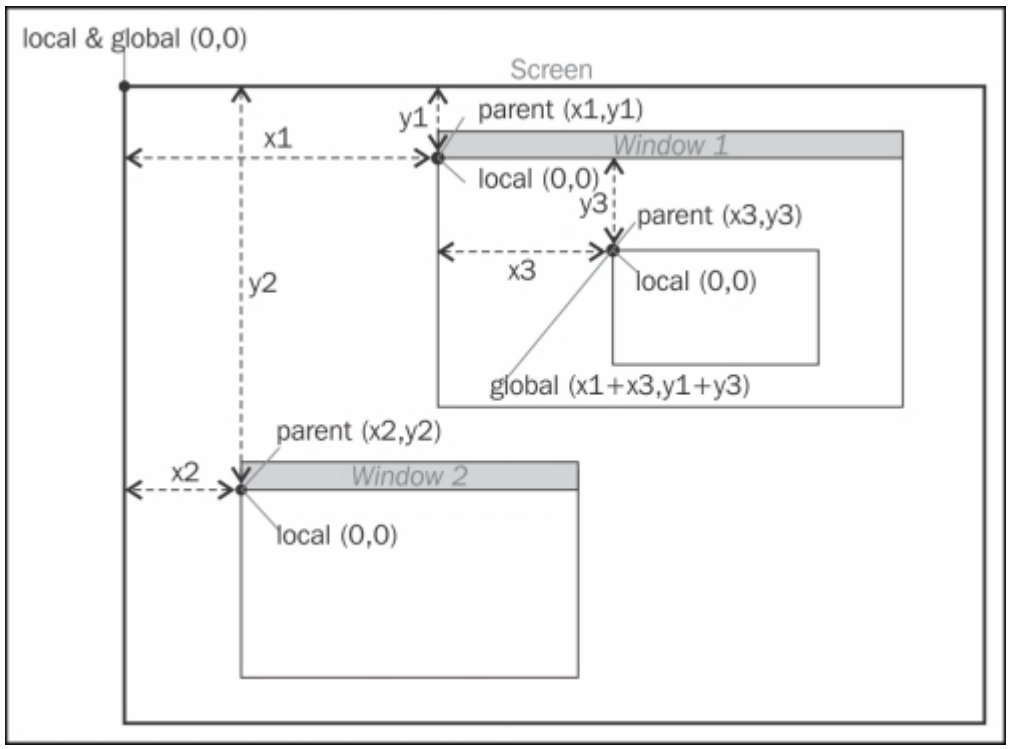
This chapter covers the JUCE `Component` class, which is the main building block for creating a **Graphical User Interface** (**GUI**) in JUCE. In this chapter we will cover the following topics:

- Creating buttons, sliders, and other components
- Responding to user interaction and changes: broadcasters and listeners
- Using other component types
- Specifying colors and using drawing operations

By the end of this chapter, you will be able to create a basic GUI and perform fundamental drawing operations within a component. You will also have the skills required to design and build more complex interfaces.

Creating buttons, sliders, and other components

The JUCE `Component` class is the base class that provides the facility to draw on the screen and intercept user interaction from pointing devices, touch-screen interaction, and keyboard input. The JUCE distribution includes a wide range of `Component` subclasses, many of which you may have encountered by exploring the JUCE Demo application in Chapter 1, *Installing JUCE and the Introjucer Application*. The JUCE coordinate system is hierarchical, starting at the computer's screen (or screens) level. This is shown in the following diagram:



Each on-screen window contains a single **parent** component within which other **child** components (or **subcomponents**) are placed (each of which may contain further child components). The top-left of the computer screen is coordinate (0, 0) with each top-left of the content of JUCE windows being at an offset from this. Each component then has its own local coordinates where its top-left starts at (0, 0) too.

In most cases you will deal with the components' coordinates relative to their parent components, but JUCE provides simple mechanisms to convert these values to be relative to other components or the main screen (that is, global coordinates). Notice in the preceding diagram that a window's top-left position does not include the title bar area.

You will now create a simple JUCE application that includes some fundamental component types. As the code for this project is going to be quite simple, we will write all our code into the header file (`.h`). This is not recommended for real-world projects except for quite small classes (or where there are other good reasons), but this will keep all the code in one place as we go through it. Also, we will split up the code into the `.h` and `.cpp` files later in the chapter.

Create a new JUCE project using the Introjucer application:

1. Choose menu item **File** | **New Project…**
2. Select **Create a Main.cpp file and a basic window** from the **Files to Auto-Generate** menu.
3. Choose where to save the project and name it `Chapter02_01`.
4. Click on the **Create…** button
5. Navigate to the **Files** panel.
6. Right-click on the file `MainComponent.cpp`, choose **Delete** from the contextual menu, and confirm.
7. Choose menu item **File** | **Save Project**.
8. Open the project in your **Integrated Development Environment** (**IDE**), either Xcode or Visual Studio.

Navigate to the `MainComponent.h` file in your IDE. The most important part of this file should look similar to this:

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

class MainContentComponent   : public Component
{
public:
  //==============================================================
  MainContentComponent();
  ~MainContentComponent();

  void paint (Graphics&);
  void resized();

  private:
  //==============================================================
  JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR
    (MainContentComponent)
};
```

Of course, we have removed the actual code from the autogenerated project by removing the `.cpp` file.

First let's make an empty window. We will remove some of the elements to simplify the code and add a function body for the constructor. Change the declaration of the `MainContentComponent` class shown as follows:

```
class MainContentComponent : public Component
{
public:
  MainContentComponent()
  {
    setSize (200, 100);
  }
};
```

Build and run the application, there should be an empty window named **MainWindow** in the center of the screen. Our JUCE application will create a window and place an instance of our `MainContentComponent` class as its content (that is, excluding the title bar). Notice our `MainContentComponent` class inherits from the `Component` class and therefore has access to a range of functions implemented by the `Component` class. The first of these is the `setSize()` function, which sets the width and height of our component.

Adding child components

Building user interfaces using components generally involves combining other components to produce composite user interfaces. The easiest way to do this is to include member variables in which to store the **child** components in the **parent** component class. For each child component that we wish to add, there are five basic steps:

1. Creating a member variable in which to store the new component.
2. Allocating a new component (either using static or dynamic memory allocation).
3. Adding the component as a child of the parent component.
4. Making the child component visible.
5. Setting the child component's size and position within the parent component.

First, we will create a button; change the code shown as follows. The preceding numbered steps are illustrated in the code comments:

```
class MainContentComponent : public Component
{
public:
  MainContentComponent()
  : button1 ("Click") // Step [2]
  {
    addAndMakeVisible (&button); // Step [3] and [4]
    setSize (200, 100);
  }

  void resized()
  {
    // Step [5]
    button1.setBounds (10, 10, getWidth()-20, getHeight()-20);
  }

private:
  TextButton button1; // Step [1]
};
```

The important parts of the preceding code are:

- An instance of the JUCE `TextButton` class was added to the `private` section of our class. This button will be statically allocated.

- The button is initialized in the constructor's initializer list using a string that sets the text that will appear on the button.
- A call to the component function `addAndMakeVisible()` is passed as a pointer to our button instance. This adds the child component to the parent component hierarchy and makes the component visible on screen.
- The component function `resized()` is overridden to position our button with an inset of 10 pixels within the parent component (this is achieved by using component functions `getWidth()` and `getHeight()` to discover the size of the parent component). This call to the `resized()` function is triggered when the parent component is resized, which in this case happens when we call the `setSize()` function in the constructor. The arguments to the `setSize()` function are in the order: width and height. The arguments to the `setBounds()` function are in the order: left, top, width, and height.

Build and run the application. Notice that the button responds as the mouse pointer hovers over the button and when the button is clicked, although the button doesn't yet do anything.

Generally, this is the most convenient method of positioning and resizing child components, even though in this example we could have easily set all the sizes in the constructor. The real power of this technique is illustrated when the parent component becomes resizable. The easiest way to do that here is to enable the resizing of the window itself. To do this, navigate to the `Main.cpp` file (which contains the boilerplate code to set up the basic application) and add the following highlighted line to the `MainWindow` constructor:

```
...
{
  setContentOwned (new MainContentComponent(), true);

  centreWithSize (getWidth(), getHeight());
  setVisible (true);
  setResizable (true, true);
}
...
```

Build and run the application and notice that the window now has a corner resizer in the bottom-right. The important thing here is that the button automatically resizes as the window size changes due to the way we implemented this above. In the call to the `setResizable()` function, the first argument sets whether the window is resizable and the second argument sets whether this is via a corner resizer (`true`) or allowing the border of the window to be dragged to resize the window (`false`).

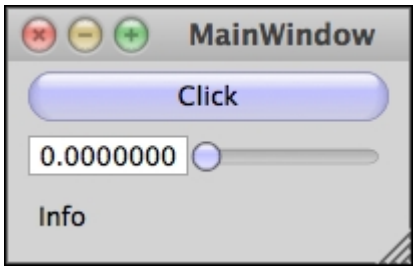
Child components may be positioned proportionally rather than with absolute or offset values. One way of achieving this is through the `setBoundsRelative()` function. In the following example you will add a slider control and a label to the component.

```
class MainContentComponent : public Component
{
public:
  MainContentComponent()
  : button1 ("Click"),
    label1 ("label1", "Info")
  {
    slider1.setRange (0.0, 100.0);
    addAndMakeVisible (&button1);
    addAndMakeVisible (&slider1);
    addAndMakeVisible (&label1);
    setSize (200, 100);
  }

  void 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);
  }

private:
  TextButton button1;
  Slider slider1;
  Label label1;
};
```

In this case, each child component is 90 percent of the width of the parent component and positioned five percent of the parent's width from the left. Each child component is 25 percent of the height of the parent, and the three components are distributed top to bottom with the button five percent of the parent's height from the top. Build and run the application, and notice that resizing the window automatically and smoothly, updates the sizes and position of the child components. The window should look similar to the following screenshot. In the next section you will intercept and respond to user interaction:



Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.