

C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# Getting started with JUCE

Leverage the power of the JUCE framework to start developing applications

**Martin Robinson**

**[PACKT]** open source\*  
PUBLISHING community experience distilled

# Getting Started With JUCE

Leverage the power of the JUCE framework to start developing applications

**Martin Robinson**



BIRMINGHAM - MUMBAI

# Getting Started With JUCE

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2013

Production Reference: 1211013

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK..

ISBN 978-1-78328-331-6

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Aniket Sawant ([aniket\\_sawant\\_photography@hotmail.com](mailto:aniket_sawant_photography@hotmail.com))

# Credits

**Author**

Martin Robinson

**Reviewers**

Michael Hetrick

Liam Lacey

Owen S. Vallis

**Acquisition Editors**

Ashwin Nair

Usha Iyer

**Lead Technical Editor**

Mohammed Fahad

**Technical Editor**

Menza Mathew

**Project Coordinator**

Suraj Bist

**Proofreader**

Clyde Jenkins

**Indexer**

Hemangini Bari

**Graphics**

Yuvraj Mannari

**Production Coordinator**

Pooja Chiplunkar

**Cover Work**

Pooja Chiplunkar

# About the Author

**Martin Robinson** is a British University Lecturer, software developer, composer, and an artist. He lectures in Music Technology with particular interests in audio software development and game audio. He gained his B.A. (Hons) Music and Music Technology from Middlesex University with first class honors, specializing in music composition. Later, he gained his M.A. Electronic Arts (with distinction) where his thesis focused on developing a system for employing artificial neural networks for controlling audio-visual systems.

His interest in computer programming developed through his music composition practice and his desire to customize systems for manipulating music and audio. He developed the UGen++ library for developing audio applications that was based on the look and feel of the SuperCollider audio programming language. Later, he developed the Plink | Plonk | Plank libraries too, for audio application development. He also develops iOS applications.

---

I would like to thank my wife Catherine and my two children, Mia and Esme, for their support, especially on writing days.

---

# About the Reviewers

**Michael Hetrick** (born in 1988) is a Ph.D. student in the Media Arts and Technology department at UC, Santa Barbara. A lifelong musician, he discovered his passion for electronic music while studying at Western Reserve Academy. At Vanderbilt University, he expanded his work into the field of video art while receiving a B.A. in Digital Media and Distribution. He went on to receive an M.A. in Electronic Music and Sound Design at UC, Santa Barbara in 2011 while doing research in chaotic synthesis under Curtis Roads, Clarence Barlow, Matthew Wright, and Marcos Novak. He is the co-owner and co-founder of Unfiltered Audio, a company dedicated to creating new software for digital musicians everywhere. His current research is focused on new paradigms for microsound. You can find his work at <http://mhetrick.com>.

**Liam Lacey** graduated from the University of the West of England, Bristol, UK in 2010 with a first-class B.Sc. (Hons) degree in Audio and Music Technology. Since then he has become the lead software developer for Bristol-based company *nu desine*, which develops the new electronic musical instrument – the **AlphaSphere**.

Liam's main interests are within the fields of music interaction and electronic musical instrument design. He recently attended the annual *New Interfaces for Music Expression* 2013 conference, where he co-authored and presented a paper on the design of the AlphaSphere. In his spare time he is also a musician, producer, composer, and performer, as well as likes getting involved in various programming projects.

Liam has been using JUCE, for more than two years, as the main library for all software that he develops, due to its ease for creating cross-platform GUI applications, and it has been the perfect framework for developing the AlphaSphere software due to JUCE's strong audio and MIDI support.

**Owen S. Vallis** is currently a professor of Music Technology at the California Institute of the Arts, Music Technology: Interaction, Intelligence, and Design program. He is a musician, artist, and scientist interested in performance, sound, and technology. As a co-founder of **Flipmu** and **The NOISE INDEX**, he explores a diverse range of projects including big data research, sound art installations, producing and composing, designing audio processors, and creating new hardware interfaces for musical performance. He received his Ph.D. in 2013 at the New Zealand School of Music, Victoria University of Wellington, and explored contemporary approaches to live computer music. During his graduate research, Owen focused on developing new musical interfaces, interactive musical agents, and large networked music ensembles. He graduated with a B.A. in Music Technology from the California Institute of the Arts in 2008.

Having lived in Toronto, Canada; Wellington, New Zealand; Tokyo, Japan; San Francisco, Nashville, and Los Angeles, Owen has been able to develop a broad and interesting cross section of musical ideologies and aesthetics. Over the past 10 years, he has worked as a research scientist for Twitter, developed multi-touch interfaces for Nokia research labs, worked for leading ribbon microphone manufacturer Royer Labs, has had musical production featured in major motion films, built a recording facility, and produced, engineered, and mixed records in Tokyo, Nashville, and Los Angeles. Owen's work has been featured in Wired, Future Music, Pitchfork, XLR8R, Processing.org, computer arts magazine, and shown at events such as NASA's Yuri's Night, Google I/O, and the New York Cutlog art festival.

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.





# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Installing JUCE and the Introjucer Application</b>	<b>7</b>
<b>Installing JUCE for Mac OS X and Windows</b>	<b>7</b>
<b>Building and running the JUCE Demo application</b>	<b>9</b>
Running the JUCE Demo application on Windows	9
Running the JUCE Demo application on Mac OS X	10
The JUCE Demo application overview	11
Customizing the look and feel	12
<b>Building and running the Introjucer application</b>	<b>13</b>
Building the Introjucer application on Windows	14
Building the Introjucer application on Mac OS X	14
Examining the JUCE Demo Introjucer project	14
<b>Creating a JUCE project with the Introjucer application</b>	<b>18</b>
<b>Documentation and other examples</b>	<b>21</b>
<b>Summary</b>	<b>22</b>
<b>Chapter 2: Building User Interfaces</b>	<b>23</b>
<b>Creating buttons, sliders, and other components</b>	<b>24</b>
Adding child components	26
<b>Responding to user interaction and changes</b>	<b>30</b>
Broadcasters and listeners	31
Filtering data entry	35
<b>Using other component types</b>	<b>37</b>
<b>Specifying colors</b>	<b>38</b>
Component color IDs	40
Setting colors using the LookAndFeel class	41

<b>Using drawing operations</b>	<b>43</b>
Intercepting mouse activity	48
<b>Configuring complex component arrangements</b>	<b>52</b>
Other component types	55
<b>Summary</b>	<b>56</b>
<b>Chapter 3: Essential Data Structures</b>	<b>57</b>
<b>Understanding the numerical types</b>	<b>58</b>
<b>Specifying and manipulating text strings</b>	<b>59</b>
Posting log messages to the console	59
String manipulation	60
<b>Measuring and displaying time</b>	<b>62</b>
Displaying and formatting time information	63
Manipulating time data	64
Measuring time	66
Accessing various special directory locations	69
Obtaining various information about files	70
Other special locations	71
Navigating directory structures	72
<b>Using dynamically allocated arrays</b>	<b>74</b>
Finding the files in a directory	76
Tokenizing strings	77
Arrays of components	77
Using the OwnedArray class	80
Other banks of controls	82
<b>Employing smart pointer classes</b>	<b>85</b>
<b>Summary</b>	<b>88</b>
<b>Chapter 4: Using Media Files</b>	<b>89</b>
<b>Using simple input and output streams</b>	<b>89</b>
Reading and writing text files	90
Reading and writing binary files	92
<b>Reading and writing image files</b>	<b>94</b>
Manipulating image data	97
<b>Playing audio files</b>	<b>100</b>
Creating a GUI to control audio file playback	101
Adding audio file playback support	103
<b>Working with the Binary Builder tool</b>	<b>110</b>
Embedding an image file using the Introjucer application	110
<b>Summary</b>	<b>112</b>

---

<b>Chapter 5: Helpful Utilities</b>	<b>113</b>
<b>Using the dynamically typed objects</b>	<b>114</b>
Using the Value class	114
Structuring hierarchical data	116
<b>Employing undo management</b>	<b>121</b>
<b>Adding XML support</b>	<b>124</b>
<b>Understanding how JUCE handles multiple threads</b>	<b>127</b>
<b>Storing application properties</b>	<b>130</b>
<b>Adding menu bar controls</b>	<b>133</b>
<b>Summary</b>	<b>137</b>
<b>Index</b>	<b>139</b>

---



# Preface

JUCE is a framework for developing cross-platform software in C++. JUCE itself comprises of a wide range of classes that solve common problems encountered when developing software systems. These include handling graphics, sound, user interaction, networks, and so on. Due to its level of audio support JUCE is popular for developing audio applications and audio plugins, although this in no means limits its use to this domain. It is relatively easy to get started with JUCE, and each JUCE class offers few surprises. At the same time, JUCE is powerful and customizable with little effort.

## What this book covers

*Chapter 1, Installing JUCE and the Introjucer Application*, guides the user through installing JUCE and covers the structure of the source code tree, including some of the useful tools available for creating JUCE projects. By the end of this chapter, the user will have installed JUCE, created a basic project using the Introjucer application and be familiar with the JUCE documentation.

*Chapter 2, Building User Interfaces*, covers the JUCE Component class, which is the main building block for creating graphical user interfaces in JUCE. By the end of this chapter, the user will be able to create basic user interfaces and perform fundamental drawing within a component. The user will also have the skills required to design and build more complex interfaces.

*Chapter 3, Essential Data Structures*, describes JUCE's important data structures, many of which could be seen as replacements for some of the standard library classes. This chapter also introduces the essential classes for JUCE development. By the end of this chapter, the user will be able to create and manipulate data in a range of JUCE's essential classes.

*Chapter 4, Using Media Files.* JUCE provides its own classes for reading and writing files and many helper classes for specific media formats. This chapter introduces the main examples of these classes. By the end of this chapter, the user will be able to manipulate a range of media files using JUCE.

*Chapter 5, Helpful Utilities.* In addition to the essential classes introduced in earlier chapters, JUCE includes a range of classes for solving common problems in application development. By the end of this chapter, the user will have an awareness of some of the additional, helpful utilities offered by JUCE.

## What you need for this book

You will need a Mac OS X or Windows computer that supports an appropriate **Integrated Development Environment (IDE)**. Any relatively recent computer should be sufficient. On Mac OS X, you should be running the Mac OS X 10.7 "Lion" operating system (or later). Most relatively recent Windows computers will support an appropriate version of the Microsoft Visual Studio IDE. Setting up the IDE for JUCE development is covered in *Chapter 1, Installing JUCE and the Introjucer Application*.

## Who this book is for

This book is for programmers with a basic grasp of C++. The examples start from a basic level, making few assumptions beyond fundamental C++ concepts. For example, not even an understanding of the C++ Standard Library is needed. Those without any experience with C++ should be able to follow and construct the examples, although may need further support to understand the fundamental concepts. Experienced programmers should also find they get to grips the JUCE library more quickly.

## Conventions

In this book, you will find a number of styles of text that distinguish among different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

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

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

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

Any command-line input or output is written as follows:

```
JUCE v2.1.2
Hello world!
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".



Tips and tricks appear like this.



## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## 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.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## **Piracy**

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## **Questions**

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.



# 1

## Installing JUCE and the Introjucer Application

This chapter guides you through installing the JUCE library, and covers the structure of its source code tree, including some of the useful tools available for creating JUCE-based projects. In this chapter we will cover the following topics:

- Installing JUCE for Mac OS X and Windows
- Building and running the JUCE Demo project
- Building and running the Introjucer application
- Creating a JUCE project with the Introjucer application

By the end of this chapter, you will have installed JUCE and created a basic project using the Introjucer application.

### Installing JUCE for Mac OS X and Windows

JUCE supports the development of C++ applications for a range of target platforms. These include Microsoft Windows, Mac OS X, iOS, Linux, and Android. In general, this book covers the development of C++ applications using JUCE for Windows and Mac OS X, but it is relatively straightforward to apply this knowledge to build applications for the other supported target platforms.

In order to compile JUCE-based code for these platforms, typically an **Integrated Development Environment (IDE)** is required. To compile code for Windows, the Microsoft Visual Studio IDE is recommended (supported variants are Microsoft Visual Studio 2008, 2010, and 2012). Microsoft Visual Studio is available to download from <http://www.microsoft.com/visualstudio> (the free Express versions are sufficient for non-commercial development). To compile code for Mac OS X or iOS, the Xcode IDE is required. Generally, the latest public version of Xcode is recommended. This can be downloaded for free from the Mac App Store from within Mac OS X.

JUCE is provided as source code (rather than prebuilt libraries) divided into discrete but interconnected **modules**. The `juce_core` module is licensed under the **Internet Systems Consortium (ISC)** license, allowing it to be used freely in commercial and open source projects. All the other JUCE modules are dual licensed. For open source development, JUCE may be licensed under the terms of the **GNU General Public License** (Version 2 or later) or the **Affero General Public License** (Version 3). JUCE may also be used for closed-source, commercial projects using separate commercial licenses for a fee. More information on JUCE licensing is available at <http://www.juce.com/documentation/commercial-licensing>.

Unless there are very specific reasons for using a particular version of JUCE, it is recommended to use the current development version available from the project's GIT repository. This version is almost always kept stable and often includes useful new features and bug fixes. The source code is available for download, using any GIT client software, at `git://github.com/julianstorer/JUCE.git` or `git://git.code.sf.net/p/juce/code`. Alternatively, the code for the current development version may be downloaded as a ZIP file from <https://github.com/julianstorer/JUCE/archive/master.zip>.

You should keep the JUCE source code in its top-level `juce` directory, but you should move this directory to a sensible location on your system that suits your workflow. The `juce` directory has the following structure (directories are shown using a trailing `/`):

```
amalgamation/  
docs/  
extras/  
juce_amalgamated.cpp  
juce_amalgamated.h  
juce_amalgamated.mm  
juce.h  
modules/  
README.txt
```

Although all of these files are important, and the actual code for the JUCE library itself is located in the `juce/modules` directory, each module is contained within its own subdirectory. For example, the `juce_core` module mentioned previously is in the `juce/modules/juce_core` directory. The remainder of this chapter examines some important projects available in the `juce/extras` directory. This directory contains a range of useful projects, in particular the JUCE Demo and the Introjucer projects.

## Building and running the JUCE Demo application

To give an overview of the features provided by JUCE, a demonstration project is included in the distribution. This is not only a good place to start, but is also a useful resource containing many examples of implementation details of classes throughout the library. This JUCE Demo project can be found in `juce/extras/JuceDemo`. The structure of this directory is typical of a JUCE project generated by the **Introjucer** application (which is covered later in the chapter).

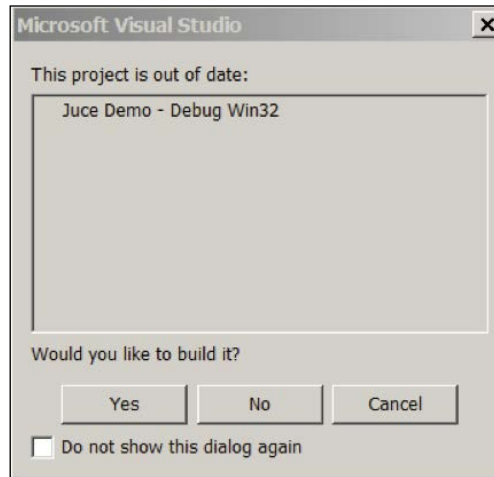
Project directory contents	Purpose
Binary Data	A directory containing any binary files, such as image and audio files, which will be embedded as code in the project
Builds	A directory containing the native platform IDE project files
Juce Demo.jucer	The Introjucer project file
JuceLibraryCode	The generic JUCE library code, configuration files, and the binary files converted to source code for inclusion in the project
Source	The project-specific source code

To build and run the JUCE Demo application, open the appropriate IDE project file from the `juce/extras/Builds` directory.

## Running the JUCE Demo application on Windows

On Windows, open the appropriate Microsoft Visual Studio Solution file. For example, using Microsoft Visual Studio 2010, this will be `juce/extras/JuceDemo/Builds/VisualStudio2010/Juce Demo.sln` (other project and solution file versions are also available for Microsoft Visual Studio 2008 and 2012).

Now, build and run the project by navigating to the menu item **Debug | Start Debugging**. You may be asked if you want to build the project first as shown in the following screenshot:



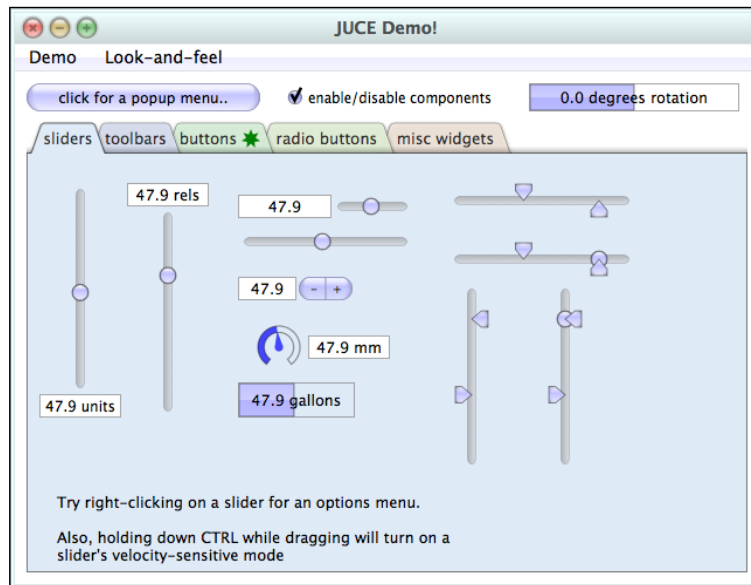
Click on **Yes**, and when this succeeds, the JUCE Demo application should appear.

## Running the JUCE Demo application on Mac OS X

On Mac OS X, open the Xcode project in: `juce/extras/JuceDemo/Builds/MacOSX/Juce Demo.xcodeproj`. To build and run the JUCE Demo application, navigate to the menu item **Product | Run**. When this succeeds, the JUCE Demo application should appear.

## The JUCE Demo application overview

The JUCE Demo application is divided into a series of demonstration pages, each illustrating a useful facet of the JUCE library. The following screenshot shows the *Widgets* demo (as it looks on Mac OS X). This is available by navigating to the menu item **Demo | Widgets**.



The Widgets demonstration shows many of the commonly needed **Graphical User Interface (GUI)** controls provided by JUCE for application development. In JUCE, these graphical elements are called **components** and this is the focus of *Chapter 2, Building User Interfaces*. There are a range of sliders, dials, buttons, text display, radio buttons, and other components, which are all customizable. There are other demonstrations available by default in the **Demo** menu, covering features such as **Graphics Rendering, Fonts and Text, Multithreading, Treeviews, Table Components, Audio, Drag-and-drop, Interprocess comms, Web Browser, and Code Editor**. There are additional demonstrations available on some platforms and when certain hardware and software is available. These are the **QuickTime, DirectShow, OpenGL, and Camera Capture** demonstrations.

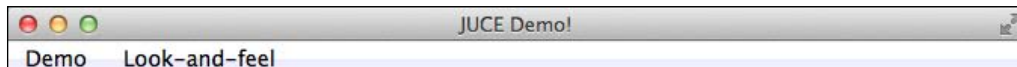


## Customizing the look and feel

By default, the JUCE Demo application uses JUCE's own window title bars, its own menu bar appearance, and its default **look and feel**. The title bars can be configured to use the native operating system appearance. The following screenshot shows the title bar of the JUCE Demo application as it appears on the Windows platform. Notice that even though the appearance of the buttons is the same as on Mac OS X, their positions should be more familiar to users on Windows.



By navigating to the menu item **Look-and-feel | Use native window title bar**, the title bar can use the standard appearance available on the operating system. The following screenshot shows the appearance of the native title bar on Mac OS X:



The default menu bar appearance, whereby the menu items appear within the application window below the title bar, should be familiar to Windows users. Of course, this is not the default location for application menus on the Mac OS X platform. Again, this can be specified as an option demonstrated in the JUCE Demo application by navigating to the menu item **Look-and-feel | Use the native OSX menu bar**. This moves the menu bar to the top of the screen, which will be more familiar to Mac OS X users. All of these options are customizable within JUCE-based code.

JUCE also provides a mechanism to customize the look and feel of many of the built-in components using its `LookAndFeel` class. This look and feel can apply to only some of the components of a particular type or globally across the application. JUCE itself, and the JUCE Demo application, come with two look and feel options: the *default* look and feel and the *old, original* (that is, "old school") look and feel. In the JUCE Demo application, this can be accessed via the **Look-and-feel** menu.

You should explore the JUCE Demo application before moving onto the next section, where you will build the Introjucer application that eases the management of multi-platform projects.

## Building and running the Introjucer application

The Introjucer application is a JUCE-based application for creating and managing multi-platform JUCE projects. The Introjucer application is able to generate the Xcode projects for Mac OS X and iOS, the Microsoft Visual Studio projects (and solutions) for Windows projects, and the project files for all the other supported platforms (and other IDEs, such as the cross-platform IDE CodeBlocks). The Introjucer application performs a number of tasks that make managing such projects much easier, such as:

- Populating all the native IDE project files with the source code files for your project
- Configuring the IDE project settings to link to the necessary libraries on the target platform
- Adding any preprocessor macros to some or all of the target IDE projects
- Adding the library and header search paths to the IDE projects
- Naming the product and adding any icon files
- Customizing the debug and release configurations (for example, code optimization settings)

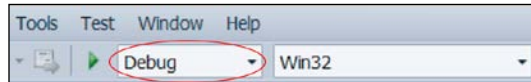
These are all helpful when setting up a project for the first time, but even more valuable when changes need to be made later in a project. Even changing the name of the product is relatively tedious if this needs to be done in several, separate projects. With the Introjucer application, most project settings can be set within the Introjucer project file itself. When saved, this will then modify the native IDE projects with any new settings. You should be aware that this would also override any changes made to the native IDE projects. Therefore, it is wise to make all the required changes within the Introjucer application.

In addition to this, the Introjucer application includes a GUI editor for arranging any GUI components. This reduces the amount of coding required for certain types of GUI development. This part of the Introjucer application generates the C++ code required to reconstruct the GUI when your application runs.

The Introjucer application is provided as source code; you will need to build it before it can be used. The source code is located in `juce/extras/Introjucer`. In a similar way to building the JUCE Demo application, there are various IDE projects available in `juce/extras/Introjucer/Builds` (understandably there are no Introjucer builds for iOS or Android). It is preferable to build the Introjucer application in its release configuration to take advantage of any code optimizations.

## Building the Introjucer application on Windows

Open the appropriate solution file in `juce/extras/Introjucer/Builds` into Microsoft Visual Studio. Change the solution configuration to from **Debug** to **Release** as shown in the following screenshot:



Now you should build the Introjucer project by navigating to the menu item **Build | Build Solution**. After this succeeds, the Introjucer application will be available at `juce/extras/Introjucer/Builds/VisualStudio2010/Release/Introjucer.exe` (or similar, if you are using a different version of Microsoft Visual Studio). At this point you should add a shortcut to your Desktop or **Start Menu**, or whatever suits your typical workflow.

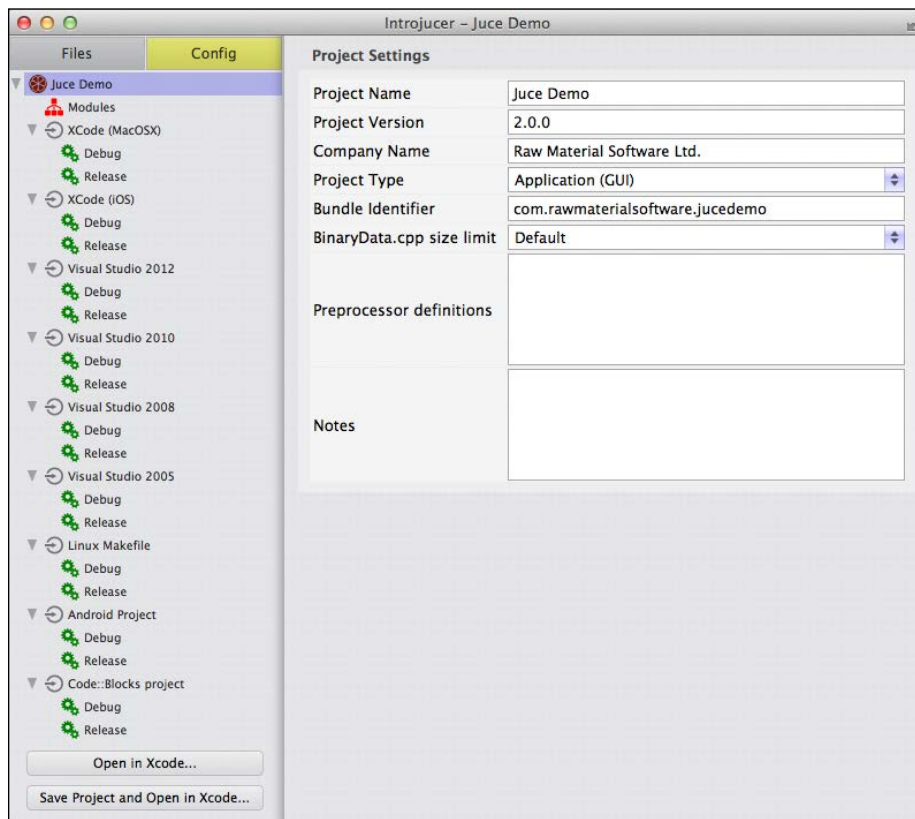
## Building the Introjucer application on Mac OS X

Open the Xcode project located at `juce/extras/Introjucer/Builds/MacOSX/TheIntrojucer.xcodeproj`. To build the Introjucer application in the release configuration, navigate to the menu item **Product | Build For | Archiving**. After this succeeds, the Introjucer application will be available at `juce/extras/Introjucer/Builds/MacOSX/build/Release/Introjucer.app`. At this point, you should add an alias to your `~/Desktop`, or whatever suits your typical workflow.

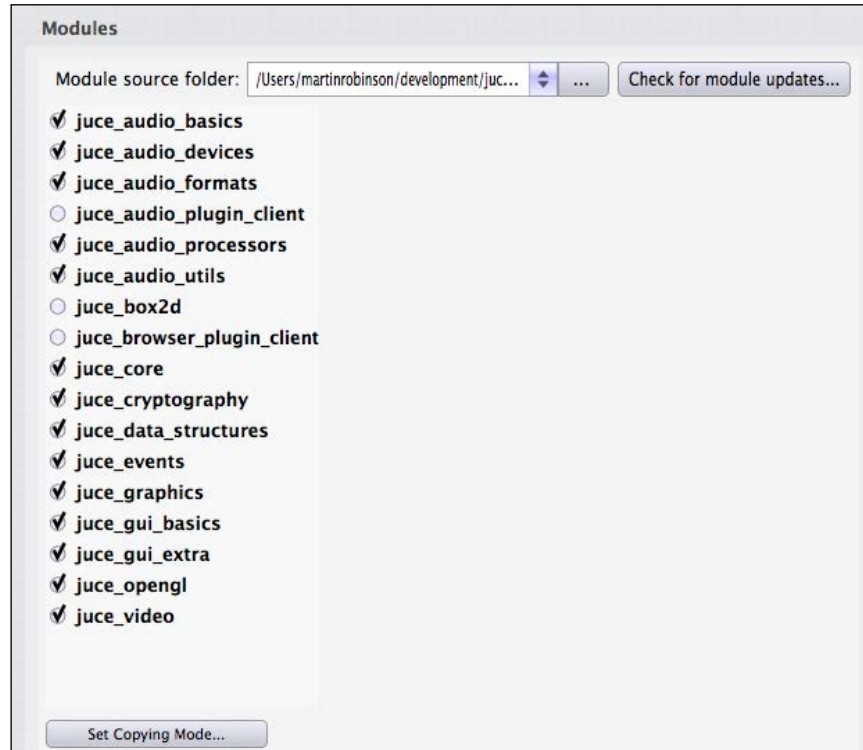
## Examining the JUCE Demo Introjucer project

To illustrate the structure and functionality of an Introjucer project, let's examine the Introjucer project for the JUCE Demo application. Open the Introjucer application you have just built on your system. In the Introjucer application, navigate to the menu item **File | Open...** and navigate to open the JUCE Demo Introjucer project file (that is, `juce/extras/JuceDemo/Juce Demo.jucer`).

An Introjucer project uses a typical **master-detail** interface, as shown in the following screenshot. On the left, or the master section, there are either the **Files** or **Config** panels, which are selectable using either the on-screen tabs or via the **View** menu. On the right, or the detail section, there are the settings associated with the selected, specific item in the master section. With the project name selected in the **Config** panel in the master section, the global settings for the whole JUCE Demo project are shown in the detail section. The **Config** panel shows the hierarchy of the project's available target builds for the different native IDEs.



In addition to these sections in the hierarchy of the **Config** panel that are concerned with the native IDE targets, there is one item named **Modules**. As mentioned previously, the JUCE codebase is divided into loosely coupled modules. Each module generally encapsulates a specific range of functionality (for example, graphics, data structures, GUI, video). The following screenshot shows the available modules and the modules that are enabled or disabled for the JUCE Demo project.

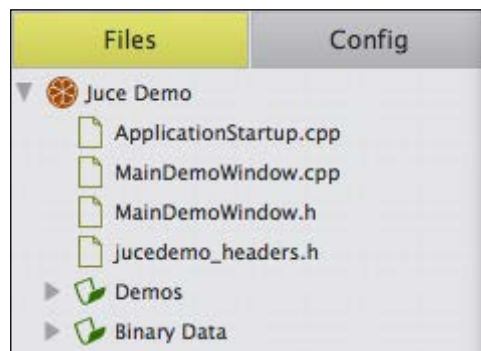


Modules may be enabled or disabled for a particular project that is based on the functionality required for the final application. For example, a simple text-editing app may not need any video or audio functionality, and the modules related to that functionality could be disabled.

Each module also has its own settings and options. In many cases, these are where there might be the option to use native libraries for certain functionality (where performance on each platform may be a high priority) or whether the cross-platform JUCE code should be used for that functionality instead (where consistency across multiple platforms is a higher priority). Each module may have a dependency on one or more other modules, in which case it will be highlighted if it has missing dependencies (and selecting the module will explain which modules need to be enabled to resolve this). To illustrate this, try turning off the checkbox for the `juce_core` module. All the other modules depend on this `juce_core` module, which, as its name suggests, provides the core functionality of the JUCE library.

Each module has a copying mode (or **Create local copy**) option. With this turned *on* (or set to **Copy the module into the project folder**), the Introjucer application will copy the source code from the JUCE source tree into the project's local project hierarchy. With this option turned *off*, the native IDE will be instructed to refer to the JUCE source files directly in the JUCE source tree. Your preference here is a matter of taste and your individual circumstances.

The left-hand **Files** panel shows the hierarchy of all the source code that will be available in the native IDEs, and the binary files (for example, images, audio, XML, ZIP) that will be transformed into cross-platform source code (and included in the native IDE projects by the Introjucer application). The top-level file structure for the JUCE Demo project is shown in the following screenshot:

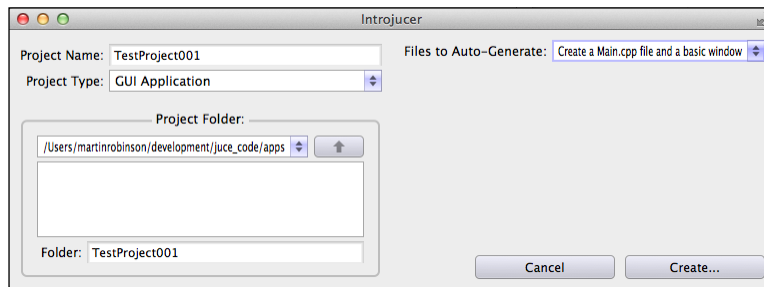


Selecting a file in the **Files** panel enables you to edit the file directly in the Introjucer application. It is currently more convenient to undertake the majority of the code editing in a native IDE that has features such as code completion, error checking, and so on.

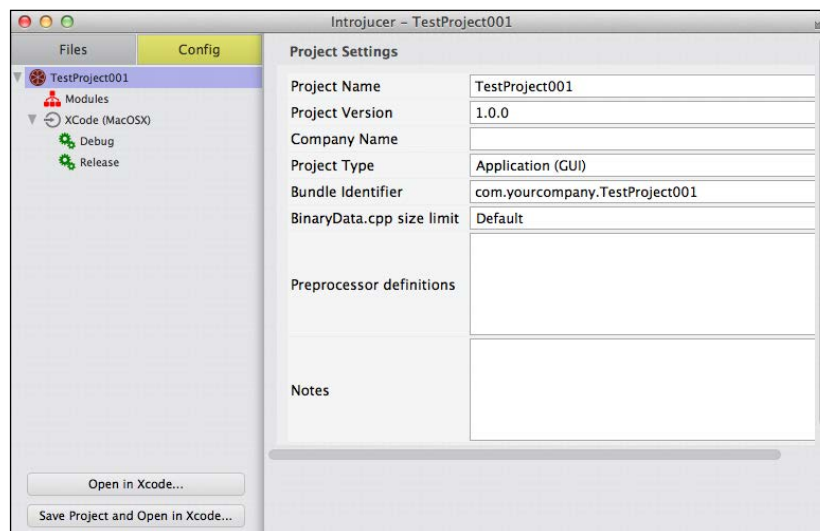
Now that we are familiar with the Introjucer application, let's use it to create a project from scratch.

## Creating a JUCE project with the Introjucer application

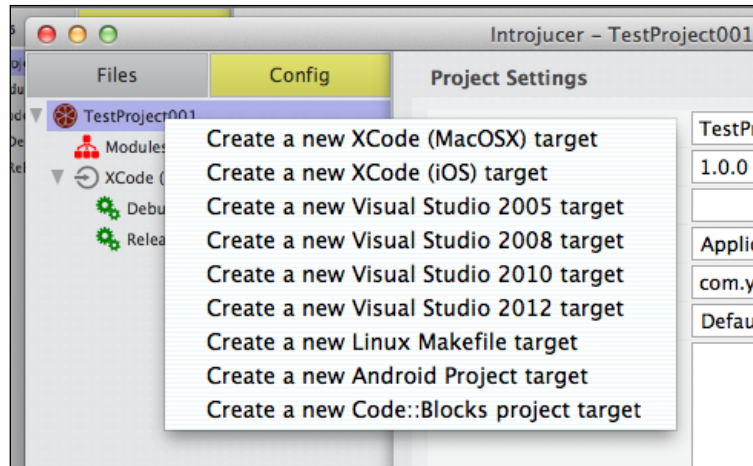
This section will guide you through creating a new Introjucer project, creating a native IDE project from this, and running your first JUCE application. First, close any open Introjucer projects by navigating to the menu item **File | Close Project**. Next, choose the menu item **File | New Project...**, and the Introjucer application will present its new project window. Using the **Project Folder** section of the window, navigate to where you would like to save the project (bearing in mind the project actually comprises a folder containing a hierarchy of code and possibly binary files). As shown in the following screenshot, name the project `TestProject001` in the **Project Name** field, and select the **Create a Main.cpp** file and a basic window option from the **Files to Auto-generate** menu:



Finally, click on the **Create...** button and a familiar Introjucer project should be presented, similar to that shown in the following screenshot:



Initially, the Introjucer application creates only one target IDE platform for the user's current platform. Right-click (on Mac OS X, press *control* and click) on the project name in the **Config** panel. This presents a range of options for adding the target platforms to the project, as shown in the following screenshot:



Select the **Files** panel and notice that the Introjucer application created three files for this basic project:

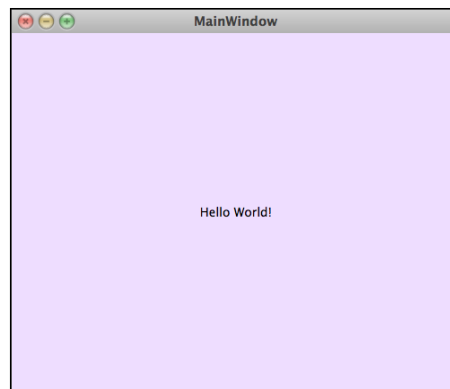
- `Main.cpp`: This manages the lifetime of the application and includes the main entry point of the application. It also includes the code to present the main application window to the user. This window in turn presents a `MainContentComponent` object in this window, which is specified in the remaining two files.
- `MainComponent.cpp`: This includes the code to draw content into this main application window. In this case, this is simply a "Hello world!" message, but could comprise a complex and hierarchical user interface.
- `MainComponent.h`: The header file for the `MainComponent.cpp` file.

It is recommended that you add any new files to the projects using this Introjucer project page. As discussed earlier, this ensures that any new files are added to all the projects for all the target platforms, rather than you having to manage this separately. In this example you will not be adding any files. Editing source files in the native IDE is not a problem even though these exact same files are used to compile on all the other platforms your project supports (that is, these files are not copied separately for each platform). You may need to be aware of some differences between compilers, but relying on the JUCE classes where possible (where this has already been taken into account) will help in this regard.

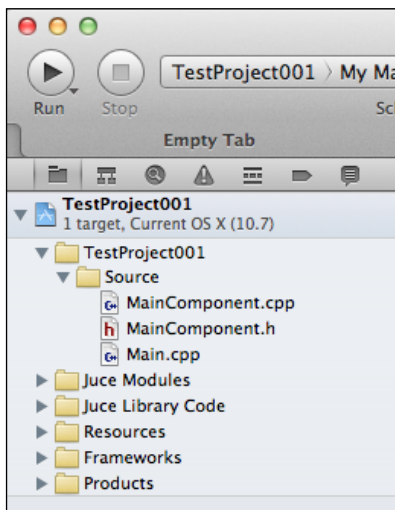


To open the project in your native IDE, first save the project by navigating to the menu item **File | Save Project**. Then, choose the appropriate option from the **File** menu to open the native project in the IDE. On Mac OS X, this menu item is **Open in Xcode....**, and on Windows it is **Open in Visual Studio....** There is also a **Menu** option that combines these two operations, and a shortcut button for it at the bottom of the **Config** panel.

Once the project is loaded into your IDE, you should build and run the project as you did with the JUCE Demo project earlier. When this succeeds, you should be presented with a window as shown in the following screenshot:



The three source files added to the project by the Introjucer application can be seen in your native IDE. The following screenshot shows the project structure in Xcode on Mac OS X. This is similar in Microsoft Visual Studio.



Edit the `MainComponent.cpp` file (by single-clicking in Xcode or double-clicking in Microsoft Visual Studio). Examine the `MainContentComponent::paint()` function. This contains four function calls to draw into the Component object's Graphics context:

- `Graphics::fillAll()`: This fills the background with a particular color
- `Graphics::setFont()`: This sets the font to a given font and size
- `Graphics::setColour()`: This sets the foreground drawing color to a particular color
- `Graphics::drawText()`: This draws some text at a specified location

Experiment with changing some of these values, and build the application again.

## Documentation and other examples

JUCE is fully documented at the following URL:

<http://www.juce.com/juce/api/>

All the JUCE classes are documented using the **Doxygen** application (<http://www.doxygen.org>), which turns specially formatted code comments into readable documentation pages. For this reason, you can also read the comments from within the JUCE source code header files if you prefer. This is sometimes more convenient, depending on your IDE, because you can navigate to the documentation easily from within the code text editor. Throughout the remainder of this book, you will be directed to the documentation for the key classes being discussed.

JUCE is used by a number of commercial developers for applications and audio plugins in particular. Some examples include:

- The **Tracktion** music production software effectively started the development of the JUCE library
- **Cycling 74's** flagship product **Max** was developed using JUCE from Version 5 onwards
- **Codex Digital** that makes products used extensively in the production of Hollywood movies
- Other important developers include **Korg**, **M-Audio**, and **TC Group**

There are many others, some of which keep their use of JUCE a secret for commercial reasons.

## Summary

This chapter has guided you through installing JUCE for your platform, and by this point you should have a good grasp of the structure of the source code tree. You should be familiar with the capabilities of JUCE through exploring the JUCE Demo project. The Introjucer application that you will have installed and used provides a basis for creating and managing projects using JUCE and the remaining chapters of this book. You will also know where to find the JUCE documentation via the JUCE website, or within the source code. In the next chapter, you will be exploring the `Component` class in more detail to create a range of user interfaces and to perform drawing operations.

# 2

## Building User Interfaces

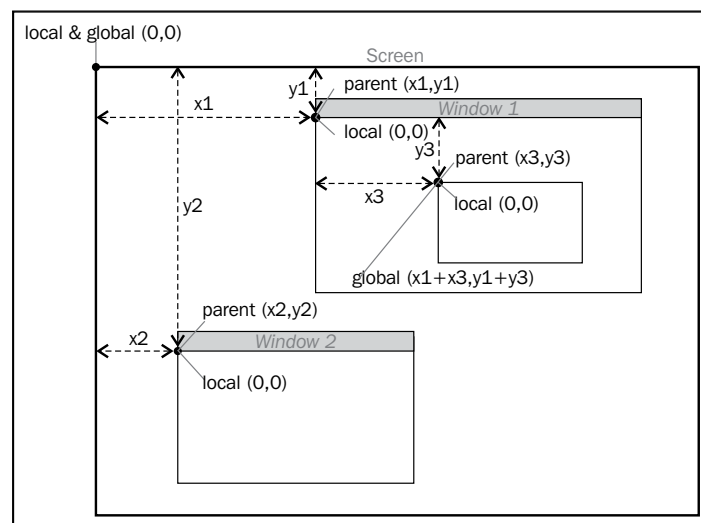
This chapter covers the JUCE `Component` class, which is the main building block for creating a **Graphical User Interface (GUI)** in JUCE. In this chapter we will cover the following topics:

- 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);
    }
};
```

```

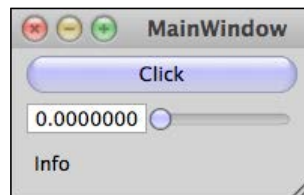
        setSize (200, 100);
    }

    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:



#### 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.

## Responding to user interaction and changes

Create a new Introjucer project named `Chapter02_02` with a basic window; this time retain all of the auto-generated files. We will now split the code from the previous section into the `MainComponent.h` and `MainComponent.cpp` files. The `MainComponent.h` file should look as follows:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void resized();

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

The `MainComponent.cpp` file should look as follows:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: button1 ("Click")
{
    slider1.setRange (0.0, 100.0);
    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);
}
```

## Broadcasters and listeners

Although the `Slider` class already contains a text box that displays the slider's value, it will be useful to examine how this communication works within JUCE. In the next example we will:

- Remove the text box from the slider
- Make the slider's value appear in the label
- Enable the slider to be zeroed by clicking on the button

To achieve this, JUCE uses the **observer** pattern widely throughout the library to enable objects to communicate. In particular, the `Component` class and the `Component` subclasses use this to notify your code when a user interface item has been clicked, their content has been changed, and so on. In JUCE, these are generally known as **listeners** (the observers) and **broadcasters** (the subjects of the observers). JUCE also makes extensive use of multiple inheritance. One area in JUCE where multiple inheritance is particularly useful is through the use of the broadcaster and listener systems. Generally, a JUCE class that supports broadcasting its state changes has a nested class called `Listener`. Thus, the `Slider` class has the `Slider::Listener` class and the `Label` class has the `Label::Listener` class. (These are often represented by classes with similar names to help support older IDEs, for example, `SliderListener` and `LabelListener` are equivalent.) The `TextButton` class is in fact a subclass of the more generic `Button` class; therefore, its listener class is `Button::Listener`. Each of these listener classes will contain a declaration of at least one **pure virtual function**. This will require our derived class to implement these functions. Listener classes may contain other regular virtual functions, meaning they may be implemented optionally. To implement these functions, first add listener classes for the button and slider as public base classes of our `MainContentComponent` class in the `MainComponent.h` file shown as follows:

```
class MainContentComponent : public Component,
                             public Button::Listener,
                             public Slider::Listener
{
    ...
}
```

Each of our user interface listeners here requires us to implement one function to respond to its changes. These are the `buttonClicked()` and `sliderValueChanged()` functions. Add these to the `public` section of our class declaration:

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

The full listing to use for the `MainComponent.cpp` file is shown as follows:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: button1 ("Zero Slider"),
  slider1 (Slider::LinearHorizontal, Slider::NoTextBox)
{
    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 two calls to add the listeners using the `addListener()` function, pass the `this` pointer (a pointer to our `MainContentComponent` instance). This adds our `MainContentComponent` instance as a listener to both the slider and the button respectively.

Although there is only one instance of each type of component, the preceding example shows the recommend way to check which component broadcasted a change, in cases where there may be many similar components (such as banks of buttons or sliders). This technique is to check the value of the pointer received by the listener function and whether this matches the address of one of the member variables. There is one thing to note on the coding style here. You may prefer to write the `if()` statement with the arguments swapped over as follows:

```
if (button == &button1)
...
```

However, the style used throughout this book is employed to cause a deliberate compiler error if you mistype the `"=="` operator as a single `"="` character. This should help avoid bugs that might be introduced by this mistake.

Components that store some kind of value such as sliders and labels may, of course, have their state set programmatically. In this case, you can control whether its listeners are notified of the change or not (you can also customize whether this is transmitted **synchronously** or **asynchronously**). This is the purpose of the `sendNotification` value (which is an enumerated constant) in the calls to the `Slider::setValue()` and `Label::setText()` functions as in the preceding code snippet. Also, you should notice that the call to the `Slider::setValue()` function in the constructor is made *after* the class has been registered as a listener. This ensures that all the components are configured correctly from the start while minimizing the duplication of code. This code makes use of the `String` class to pass text to the label, to convert text to numerical values, and vice versa. The `String` class will be explored in more detail in the next chapter, but for now, we will limit the usage of the `String` class to these basic operations. The text box is removed from the slider by initializing the slider in the initializer list with a slider style and text box style. In this case, the initializer `slider1 (Slider::LinearHorizontal, Slider::NoTextBox)` specifies a horizontal slider and that no text box should be attached.

Finally, should we want to set the value of the slider to something specific; we can make the label editable and transmit any changes typed into the label to the slider. Make another new Introjucer project and name it `Chapter02_03`. Add the `Label::Listener` class to the base classes of our `MainContentComponent` class in the header file:

```
class MainContentComponent : public Component,
                             public Button::Listener,
                             public Slider::Listener,
                             public Label::Listener
{
    ...
}
```

Add the `Label::Listener` function that responds to the label changes, also in the header file:

```
...
void labelTextChanged (Label* label);
...
```

Update the constructor in the `MainComponent.cpp` file to further configure the label:

```
MainContentComponent::MainContentComponent()
: button1 ("Zero Slider"),
  slider1 (Slider::LinearHorizontal, Slider::NoTextBox)
{
    slider1.setRange (0.0, 100.0);
    label1.setEditable (true);

    slider1.addListener (this);
    button1.addListener (this);
    label1.addListener (this);

    slider1.setValue (100.0, sendNotification);

    addAndMakeVisible (&button1);
    addAndMakeVisible (&slider1);
    addAndMakeVisible (&label1);

    setSize (200, 100);
}
```

Here, the label is set to be editable with a single click and our class registers itself as a listener for the label. Lastly, add the implementation for the `labelTextChanged()` function to the `MainComponent.cpp` file:

```
void MainContentComponent::labelTextChanged (Label* label)
{
    if (&label1 == label) {
        slider1.setValue (label1.getText().getDoubleValue(),
                          sendNotification);
    }
}
```

Build and run the application to test this functionality. There are some problems:

- The slider correctly clips values typed into the label that are outside the range of the slider, but the text in the label still remains if these values are outside the range
- The label allows non-numerical characters to be typed in (although these are usefully resolved to zero)

## Filtering data entry

The first issue mentioned above is straightforward, and this is to convert the slider's value back to text and use this to set the label content. This time we use the `dontSendNotification` value, because we want to avoid an infinite loop whereby each component would broadcast a message that causes a change that would in turn cause a message to be broadcasted and so on:

```
if (&label1 == label)
{
    slider1.setValue (label1.getText().getDoubleValue(),
                      sendNotification);
    label1.setText (String (slider1.getValue()),
                    dontSendNotification);
}
```



The second issue requires a filter to allow only certain characters. Here, you need access to the label's internal `TextEditor` object. To do this, you could create a custom label class by inheriting from the `Label` class and implementing the `editorShown()` virtual function. Add this small class to the `MainComponent.h` file above the `MainContentComponent` class declaration (although to reuse this class across a number of components in your application, it may be better to place this code in a separate file):

```
class NumericalLabel : public Label
{
public:
    void editorShown (TextEditor* editor)
    {
        editor->setInputRestrictions (0, "-0123456789.");
    }
};
```

Because the text editor is just about to be shown, this function is called by the label, and at that point you can set the text editor's input restrictions using its `setInputRestrictions()` function. The two arguments are: length and allowable characters. The zero length means there is no restriction on length and the allowable characters in this case include all the digits, the minus sign and the period. (In fact you could omit the minus sign to disallow negative numbers and omit the period if you wanted to allow integer values only.) To use this class in place of the built-in `Label` class simply replace this class name in the member variable list for our `MainContentComponent` class as shown highlighted:

```
...
private:
    TextButton button1;
    Slider slider1;
    NumericalLabel label1;
...
```

Hopefully, by this point you can see that JUCE classes provide a useful range of core functionality while allowing customizations with relative ease.

## Using other component types

There are many other built-in component types and variations on the sliders and buttons already seen. In the previous section we used the default horizontal slider, but the `Slider` class is very flexible, as illustrated by the Widget demo page of the JUCE Demo application. The sliders can adopt a rotary-type control, have minimum and maximum ranges, and warp the numerical track to adopt non-linear behavior. Similarly, buttons can adopt different styles such as toggle buttons, buttons that use images, and so on. The following example illustrates a toggle-type button that changes the style of two sliders. Create a new Introjucer project named `Chapter02_04`, and use the following code:

- **MainComponent.h:**

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

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

    void buttonClicked (Button* button);

private:
    Slider slider1;
    Slider slider2;
    ToggleButton toggle1;
};
#endif
```

- **MainComponent.cpp:**

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: slider1 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  slider2 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  toggle1 ("Slider style: Linear Bar")
{
    slider1.setColour (Slider::thumbColourId, Colours::red);
    toggle1.addListener (this);
```

```
        addAndMakeVisible (&slider1);
        addAndMakeVisible (&slider2);
        addAndMakeVisible (&toggle1);

        setSize (400, 200);
    }

    void MainContentComponent::resized()
    {
        slider1.setBounds (10, 10, getWidth() - 20, 20);
        slider2.setBounds (10, 40, getWidth() - 20, 20);
        toggle1.setBounds (10, 70, getWidth() - 20, 20);
    }

    void MainContentComponent::buttonClicked (Button* button)
    {
        if (&toggle1 == button)
        {
            if (toggle1.getToggleState()) {
                slider1.setSliderStyle (Slider::LinearBar);
                slider2.setSliderStyle (Slider::LinearBar);
            } else {
                slider1.setSliderStyle (Slider::LinearHorizontal);
                slider2.setSliderStyle (Slider::LinearHorizontal);
            }
        }
    }
}
```

This example uses a `ToggleButton` object and checks its toggle state in the `buttonClicked()` function using the `getToggleState()` function. One obvious customization yet to be discussed is changing the colors of the various elements within the built-in components. This will be covered in the next section.

## Specifying colors

Colors in JUCE are handled by the `Colour` and `Colours` classes (*note the British spelling of these two class names*):

- The `Colour` class stores a 32-bit color with 8-bit alpha, red, green, and blue values (**ARGB**). A `Colour` object may be initialized from other formats (for example, using floating point values, or values in the **HSV** format).
- The `Colour` class includes a number of utilities for creating new colors from existing ones, for example, by modifying the alpha channel, changing only the brightness or finding a suitable contrasting color.

- The `Colours` class is a collection of static `Colour` instances (for example, `Colour::red`, `Colour::cyan`). These are based broadly on the naming scheme of colors in the **HyperText Markup Language (HTML)** standard.

For example, the following code snippet illustrates several different ways of creating the same "red" color:

```
Colour red1 = Colours::red;           // using Colours
Colour red2 = Colour (0xffff0000);    // using hexadecimal ARGB
Colour red3 = Colour (255, 0, 0);     // using 8-bit RGB values
Colour red4 = Colour::fromFloatRGBA (1.f, 0.f, 0.f, 1.f); // float
Colour red5 = Colour::fromHSV (0.f, 1.f, 1.f, 1.f);    // HSV
```

Component classes employ an ID system to refer to the various colors they use for different purposes (background, border, text, and so on). To use these colors to change the appearance of a component, the `Component::setColour()` function is used:

```
void setColour (int colourId, Colour newColour);
```

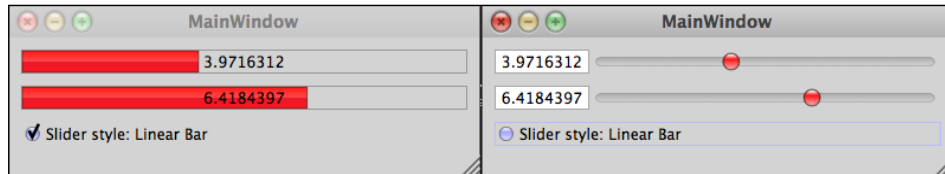
For example, to change the color of a slider's thumb (which is the draggable part), the ID is the `Slider::thumbColourId` constant (this too changes the fill color that represents the slider's value when the slider style is set to the `Slider::LinearBar` constant). You can test this in the `Chapter02_04` project by adding the following highlighted lines to the constructor:

```
MainContentComponent::MainContentComponent()
: slider1 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  slider2 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  toggle1 ("Slider style: Linear Bar")
{
    slider1.setColour (Slider::thumbColourId, Colours::red);
    slider2.setColour (Slider::thumbColourId, Colours::red);
    toggle1.addListener (this);

    addAndMakeVisible (&slider1);
    addAndMakeVisible (&slider2);
    addAndMakeVisible (&toggle1);

    setSize (400, 200);
}
```

The final look of this application showing both types of slider is shown in the following screenshot:



## Component color IDs

Many built-in components define their own color ID constants; the most useful are:

- `Slider::backgroundColourId`
- `Slider::thumbColourId`
- `Slider::trackColourId`
- `Slider::rotarySliderFillColourId`
- `Slider::rotarySliderOutlineColourId`
- `Slider::textBoxTextColourId`
- `Slider::textBoxBackgroundColourId`
- `Slider::textBoxHighlightColourId`
- `Slider::textBoxOutlineColourId`
- `Label::backgroundColourId`
- `Label::textColourId`
- `Label::outlineColourId`
- `ToggleButton::textColourId`
- `TextButton::buttonColourId`
- `TextButton::buttonOnColourId`
- `TextButton::textColourOffId`
- `TextButton::textColourOnId`

Each of these enumerated constants is defined in the class in which they are used. There are many others for each of the component types.

## Setting colors using the LookAndFeel class

If you have many controls and want to set unified colors for all of them, then it is likely to be more convenient to set the color at some other point in the component hierarchy. This is one purpose of the JUCE `LookAndFeel` class. This was seen briefly in *Chapter 1, Installing JUCE and the Introjucer Application* where the different styles of the various widgets can be selected by using a different look and feel. If this is to be a global change across the whole application then the best place to put this change is likely to be in the initialization code. To try this, remove the following two lines of code from your project, which were added in the previous step:

```
slider1.setColour (Slider::thumbColourId, Colours::red);  
slider2.setColour (Slider::thumbColourId, Colours::red);
```

Navigate to the `Main.cpp` file. Now add the following lines to the `initialise()` function (*again notice the British spelling*).

```
void initialise (const String& commandLine)  
{  
    LookAndFeel& lnf = LookAndFeel::getDefaultLookAndFeel();  
    lnf.setColour (Slider::thumbColourId, Colours::red);  
    mainWindow = new MainWindow();  
}
```

It should be clear that an extended list of colors could be configured at this point to customize the application's appearance. Another technique, that again uses the `LookAndFeel` class, is to inherit from the default `LookAndFeel` class and update colors in this derived class. Setting a particular look and feel for a component affects all child components in its hierarchy. Therefore, this method would allow you to set colors selectively in different parts of an application. A solution that uses this method is shown as follows, with the important parts highlighted:

### The MainComponent.h file:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

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

    void buttonClicked (Button* button);

    class AltLookAndFeel : public LookAndFeel
    {
    public:
        AltLookAndFeel()
        {
            setColour (Slider::thumbColourId, Colours::red);
        }
    };

private:
    Slider slider1;
    Slider slider2;
    ToggleButton toggle1;
    AltLookAndFeel altLookAndFeel;
};
#endif
```

In the MainComponent.cpp file only the constructor needs updating:

```
MainContentComponent::MainContentComponent()
: slider1 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  slider2 (Slider::LinearHorizontal, Slider::TextBoxLeft),
  toggle1 ("Slider style: Linear Bar")
{
    setLookAndFeel (&altLookAndFeel);
    toggle1.addListener (this);

    addAndMakeVisible (&slider1);
    addAndMakeVisible (&slider2);
    addAndMakeVisible (&toggle1);

    setSize (400, 200);
}
```

Here, we create a nested class `AltLookAndFeel` that is based on the default `LookAndFeel` class. This is defined as a nested class, because we need to only refer to it from within a `MainContentComponent` instance. It might be more appropriate to define this class outside the `MainContentComponent` class if the `AltLookAndFeel` becomes a more extended class or needs to be reused by other component classes that we write.

In the `AltLookAndFeel` constructor, we set the color of the slider thumb. Finally, we set the look and feel for the `MainContentComponent` class in its constructor. There are clearly many other possible techniques using this handful of tools, and the exact approach is heavily dependent on the specific application features being developed. It is important to note that the `LookAndFeel` class not only deals with colors, but also more broadly enables you to configure the exact way in which certain user interface elements are drawn. Not only can you change the color of the slider thumb, you can change its radius (by overriding the `LookAndFeel::getSliderThumbRadius()` function) or even change its shape altogether (by overriding the `LookAndFeel::drawLinearSliderThumb()` function).

## Using drawing operations

Although it is advisable to use the built-in components if possible, there are occasions where you may need or wish to create a completely new custom component. This may be to perform some specific drawing tasks or a unique user interface item. JUCE also handles this elegantly.

First, create a new Introjucer project and name it `Chapter02_05`. To perform drawing tasks in a component, you should override the `Component::paint()` function. Change the contents of the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void paint (Graphics& g);
};
#endif
```



Change the contents of the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    setSize (200, 200);
}

void MainContentComponent::paint (Graphics& g)
{
    g.fillAll (Colours::cornflowerblue);
}
```

Build and run the application to see the resulting empty window filled with a blue color.

The `paint()` function is called when the component needs to redraw itself. This might be due to the component having been resized (which of course you can try out using the corner resizer), or specific calls to invalidate the display (for example, the component displays visual representation of a value and this is no longer the currently stored value). The `paint()` function is passed a reference to a `Graphics` object. It is this `Graphics` object that you instruct to perform your drawing tasks. The `Graphics::fillAll()` function used in the code above should be self-explanatory: it fills the entire component with the specified color. The `Graphics` object can draw rectangles, ellipses, rounded rectangles, lines (in various styles), curves, text (with numerous shortcuts for fitting or truncating text within particular areas) and images.

The next example illustrates drawing a collection of random rectangles using random colors. Change the `paint()` function in the `MainComponent.cpp` file to:

```
void MainContentComponent::paint (Graphics& g)
{
    Random& r (Random::getSystemRandom());
    g.fillAll (Colours::cornflowerblue);

    for (int i = 0; i < 20; ++i) {
        g.setColour (Colour (r.nextFloat(),
                               r.nextFloat(),
                               r.nextFloat(),
                               r.nextFloat()));

        const int width = r.nextInt (getWidth() / 4);
        const int height = r.nextInt (getHeight() / 4);
        const int left = r.nextInt (getWidth() - width);
        const int top = r.nextInt (getHeight() - height);
```

```
        g.fillRect (left, top, width, height);  
    }  
}
```

This makes use of multiple calls to the JUCE random number generator class `Random`. This is a convenient class that allows the generation of pseudo-random integers and floating-point numbers. You can make your own instance of a `Random` object (which is recommended if your application uses random numbers in multiple threads), but here we simply take a copy of a reference to a global "system" `Random` object (using the `Random::getSystemRandom()` function) and use it multiple times. Here, we fill the component with a blue background and generate 20 rectangles. The color is generated from randomly generated floating point ARGB values. The call to the `Graphics::setColour()` function sets the current drawing color that will be employed by subsequent drawing commands. A randomly generated rectangle is also created by first choosing width and height (each being a maximum value of one-quarter of the parent component's width and height respectively). Then the position of the rectangle is randomly selected; again this is done using the parent component's width and height but this time subtracting the width and height of our random rectangle to ensure its right and bottom edges are not off-screen. As mentioned previously, the `paint()` function is called each time the component needs to be redrawn. This means we will get a completely new set of random rectangles as the component is resized.

Changing the drawing command to `fillEllipse()` rather than `fillRect()` draws a collection of ellipses instead. Lines can be drawn in various ways. Change the `paint()` function as follows:

```
void MainContentComponent::paint (Graphics& g)  
{  
    Random& r (Random::getSystemRandom());  
    g.fillAll (Colours::cornflowerblue);  
  
    const float lineThickness = r.nextFloat() * 5.f + 1.f;  
  
    for (int i = 0; i < 20; ++i) {  
        g.setColour (Colour (r.nextFloat(),  
                             r.nextFloat(),  
                             r.nextFloat(),  
                             r.nextFloat()));  
  
        const float startX = r.nextFloat() * getWidth();  
        const float startY = r.nextFloat() * getHeight();  
        const float endX = r.nextFloat() * getWidth();  
        const float endY = r.nextFloat() * getHeight();
```

```
        g.drawLine (startX, startY,  
                    endX, endY,  
                    lineThickness);  
    }  
}
```

Here, we choose a random line thickness (between one and six pixels wide) before the `for()` loop and use it for each line. The start and end positions of the lines are also randomly generated. To draw a continuous line there are a number of options, you could:

- store the last end point of the line and use this as the start point of the next line; or
- use a `JUCE Path` object to build a series of line drawing commands and draw the path in one pass.

The first solution would be something like this:

```
void MainContentComponent::paint (Graphics& g)  
{  
    Random& r (Random::getSystemRandom());  
    g.fillAll (Colours::cornflowerblue);  
  
    const float lineThickness = r.nextFloat() * 5.f + 1.f;  
  
    float x1 = r.nextFloat() * getWidth();  
    float y1 = r.nextFloat() * getHeight();  
  
    for (int i = 0; i < 20; ++i) {  
        g.setColour (Colour (r.nextFloat(),  
                             r.nextFloat(),  
                             r.nextFloat(),  
                             r.nextFloat()));  
  
        const float x2 = r.nextFloat() * getWidth();  
        const float y2 = r.nextFloat() * getHeight();  
  
        g.drawLine (x1, y1, x2, y2, lineThickness);  
        x1 = x2;  
        y1 = y2;  
    }  
}
```

The second option is slightly different; in particular, each of the lines that make up the path must be same color:

```
void MainContentComponent::paint (Graphics& g)
{
    Random& r (Random::getSystemRandom());
    g.fillAll (Colours::cornflowerblue);

    Path path;
    path.startNewSubPath (r.nextFloat() * getWidth(),
                          r.nextFloat() * getHeight());

    for (int i = 0; i < 20; ++i) {
        path.lineTo (r.nextFloat() * getWidth(),
                    r.nextFloat() * getHeight());
    }

    g.setColour (Colour (r.nextFloat(),
                        r.nextFloat(),
                        r.nextFloat(),
                        r.nextFloat()));

    const float lineThickness = r.nextFloat() * 5.f + 1.f;
    g.strokePath (path, PathStrokeType (lineThickness));
}
```

Here the path is created before the `for()` loop and each iteration of the loop adds a line segment to the path. These two approaches to line drawing clearly suit different applications. The path drawing technique is heavily customizable, in particular:

- The joints at the corners of the line segments can be customized with the `PathStrokeType` class (for example, to make the corners slightly rounded).
- The lines need not be straight: they can be Bezier curves.
- The path may include other fundamental shapes such as rectangles, ellipses, stars, arrows and so on.

In addition to these line drawing commands, there are accelerated functions specifically for drawing horizontal and vertical lines (that is, non-diagonal). These are the `Graphics::drawVerticalLine()` and `Graphics::drawHorizontalLine()` functions.

## Intercepting mouse activity

To help your component respond to mouse interaction, the `Component` class has six important callback functions that you can override:

- `mouseenter()`: Called when the mouse pointer enters the bounds of this component and the mouse buttons are *up*.
- `mousemove()`: Called when the mouse pointer moves within the bounds of this component and the mouse buttons are *up*. A `mouseenter()` callback will always have been received first.
- `mousedown()`: Called when one or more mouse buttons are pressed while the mouse pointer is over this component. A `mouseenter()` callback will always have been received first and it is highly likely one or more `mousemove()` callbacks will have been received too.
- `mousedown()`: Called when the mouse pointer is moved following a `mousedown()` callback on this component. The position of the mouse pointer may be outside the bounds of the component.
- `mouseup()`: Called when the mouse button is released following a `mousedown()` callback on this component (the mouse pointer will not necessarily be over this component at this time).
- `mouseleave()`: Called when the mouse pointer leaves the bounds of this component when the mouse buttons are *up* and after a `mouseup()` callback if the user has clicked on this component (even if the mouse pointer exited the bounds of this component some time ago).

In each of these cases, the callbacks are passed a reference to a `MouseEvent` object that can provide information about the current state of the mouse (where it was at the time of the event, when the event occurred, which modifier keys on the keyboard were down, which mouse buttons were down, and so on). In fact, although these classes and function names refer to the "mouse" this system can handle multi-touch events and the `MouseEvent` object can be asked which "finger" was involved in such cases (for example, on the iOS platform).

To experiment with these callbacks, create a new Introjucer project and name it `Chapter02_06`. Use the following code for this project.

The `MainComponent.h` file declares the class with its various member functions and data:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void paint (Graphics& g);

    void mouseEnter (const MouseEvent& event);
    void mouseMove (const MouseEvent& event);
    void mouseDown (const MouseEvent& event);
    void mouseDrag (const MouseEvent& event);
    void mouseUp (const MouseEvent& event);
    void mouseExit (const MouseEvent& event);

    void handleMouse (const MouseEvent& event);

private:
    String text;
    int x, y;
};
#endif
```

The `MainComponent.cpp` file should contain the following code. First, add the constructor and the `paint()` function. The `paint()` function draws a yellow circle at the mouse position and some text showing the current phase of the mouse interaction:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: x (0), y (0)
{
    setSize (200, 200);
}

void MainContentComponent::paint (Graphics& g)
{
```

```
g.fillAll (Colours::cornflowerblue);
g.setColour (Colours::yellowgreen);
g.setFont (Font (24));
g.drawText (text, 0, 0, getWidth(), getHeight(),
            Justification::centred, false);
g.setColour (Colours::yellow);
const float radius = 10.f;
g.fillEllipse (x - radius, y - radius,
               radius * 2.f, radius * 2.f);
}
```

Then add the mouse event callbacks and our `handleMouse()` function described as follows. We store the coordinates of the mouse callbacks with reference to our component and store a `String` object based on the type of callback (mouse down, up, move, and so on). Because the storage of the coordinates is the same in each case, we use the `handleMouse()` function, which stores the coordinates from the `MouseEvent` object in our class member variables `x` and `y`, and pass this `MouseEvent` object from the callbacks. To ensure that the component redraws itself, we must call the `Component::repaint()` function.

```
void MainContentComponent::mouseEnter (const MouseEvent& event)
{
    text = "mouse enter";
    handleMouse (event);
}

void MainContentComponent::mouseMove (const MouseEvent& event)
{
    text = "mouse move";
    handleMouse (event);
}

void MainContentComponent::mouseDown (const MouseEvent& event)
{
    text = "mouse down";
    handleMouse (event);
}

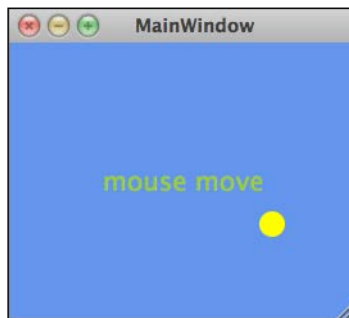
void MainContentComponent::mouseDrag (const MouseEvent& event)
{
    text = "mouse drag";
    handleMouse (event);
}
```

```
void MainContentComponent::mouseUp (const MouseEvent& event)
{
    text = "mouse up";
    handleMouse (event);
}

void MainContentComponent::mouseExit (const MouseEvent& event)
{
    text = "mouse exit";
    handleMouse (event);
}

void MainContentComponent::handleMouse (const MouseEvent& event)
{
    x = event.x;
    y = event.y;
    repaint();
}
```

As shown in the following screenshot, the result is a yellow circle that sits under our mouse pointer and a text message in the center of the window that gives feedback as to the type of mouse event most recently received:

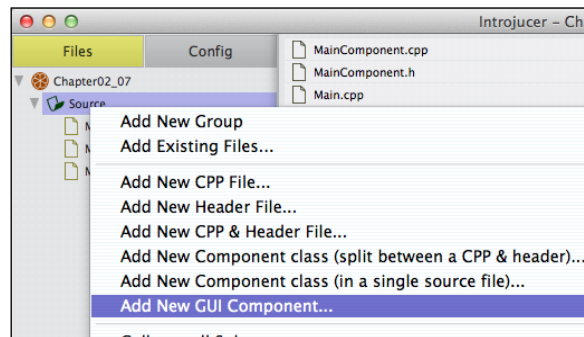




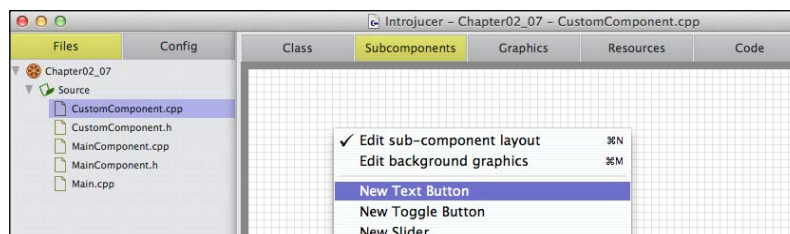
## Configuring complex component arrangements

JUCE makes it straightforward to create custom components either by combining several built-in components or through providing an effective means of interacting with pointing devices combined with a range of fundamental drawing commands. In addition to this, the Introjucer application provides a graphical editor for laying out custom components. This will then autogenerate the code required to rebuild this interface in your application. Create a new Introjucer project as earlier, with a basic window, and name it `Chapter02_07`.

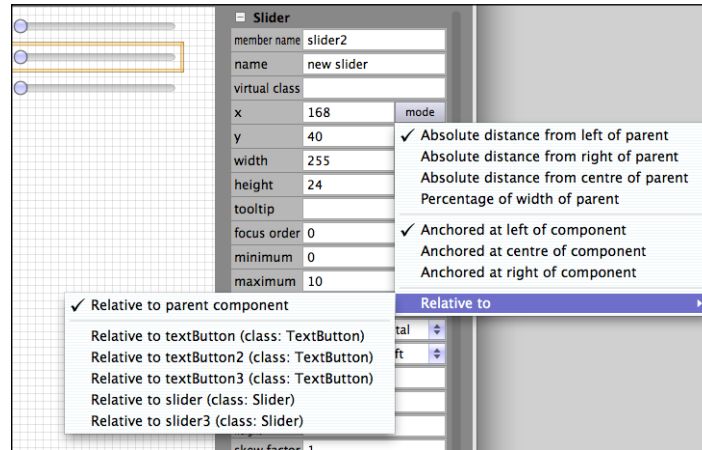
Switch to the **Files** panel, right-click (on the Mac, press *control* and click) on the **Source** folder in the hierarchy, and select **Add New GUI Component...** from the contextual menu, as shown in the following screenshot:



You will be asked to name the header file, which also names the corresponding `.cpp` file. Name the header file `CustomComponent.h`. When you select a `.cpp` file created in this way, you are offered several ways of editing the file. In particular you can add child components, add drawing commands, or you can edit the code directly. Select the `CustomComponent.cpp` file, as shown in the following screenshot:



In the **Subcomponents** panel, you can right-click on the grid to add one of the several built-in component types. Add in a few buttons and sliders. Each of these can be edited when selected using the properties on the right-hand side of the window. What is particularly useful here is the ability to set complex rules about the positioning of the components relative to each other and the parent component. Some of the options for this are visible in the following screenshot:



Because the Introjucer application generates C++ code, it should be clear that these options are clearly available programmatically. For some tasks, especially complex GUIs, using the GUI editor may be more convenient. It is also a useful way of discovering features available in the various component classes and the corresponding code to enable and control these features.

Before opening the project in your IDE, select the **Class** panel (using the tab to the left of the **Subcomponents** tab) and change the **class name** from `NewComponent` to `CustomComponent` (to match the filenames of the code). Save the Introjucer project and open its IDE project for your platform. You need make only a few minor modifications to load this auto-generated code into your `MainContentComponent` class. Change the `MainComponent.h` file as follows:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
```

```
public:
    MainContentComponent ();

private:
    CustomComponent custom;
};
#endif
```

Then, change the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent ()
{
    addAndMakeVisible (&custom);
    setSize (custom.getWidth(), custom.getHeight());
}
```

This allocates a `CustomComponent` object and makes it fill the bounds of the `MainContentComponent` object. Build and run the application, and you should see whatever user interface you designed in the Introjucer application's GUI editor. The Introjucer application takes special control of the source files for these autogenerated GUI controls. Take a look in the `CustomComponent.h` and `CustomComponent.cpp` files. There will be some code you recognize from earlier in this chapter (one major difference is that the Introjucer application generates code to allocate the subcomponent classes dynamically, rather than using static allocation as we have done here). You must be very careful when editing code in these autogenerated GUI files, because loading the project back into the Introjucer application may overwrite some of your changes (which doesn't happen with regular code files). The Introjucer application identifies areas where you *may* make changes using specially tagged opening and closing comments. For example, this is the end of a typical autogenerated component constructor:

```
...
    // [UserPreSize]
    // [/UserPreSize]

    setSize (600, 400);

    // [Constructor] You can add your own custom stuff here..
    // [/Constructor]
}
```

You *may* make changes and add code in between the opening `[UserPreSize]` tag and closing `[/UserPreSize]` tag and between the opening `[Constructor]` tag and closing `[/Constructor]` tag. In fact you can make edits between any of these opening and closing tags *but not anywhere else*. Doing so risks your changes being deleted if and when the Introjucer project is next saved to disk. This applies if you add another build target, add another GUI component, add other files to the Introjucer project, and where you explicitly save the project in the Introjucer application.

## Other component types

JUCE comprises a wide range of other component types for particular tasks. Many of these will be familiar, as similar controls are available within many operating systems and other GUI frameworks. In particular:

- **Buttons:** There are several button types, including buttons that can be created using image files and other shapes (for example, `ImageButton`, and `ShapeButton` classes); there is a `ToolBarButton` class that can be used to create toolbars.
- **Menus:** There is a `PopupMenu` class (for issuing commands) and a `ComboBox` class (for selecting items).
- **Layout:** There are various classes for organizing other components including a `TabbedComponent` class (for creating tabbed pages), a `Viewport` class (for creating scrollable content), a `TableListBox` class (for creating tables), and a `TreeView` class (for organizing content in to a hierarchical structure).
- **File browsers:** There are various ways of displaying and accessing file directory structures including the `FileChooser`, `FileNameComponent`, and `FileTreeComponent` classes.
- **Text editors:** There is a general-purpose `TextEditor` class, and a `CodeEditorComponent` for displaying and editing code.

Most of source code for these components can be found in `juce/modules/juce_gui_basics` with some additional classes being found in `juce/modules/juce_gui_extra`. All classes are documented in the online documentation. An alphabetical list of all classes can be found here:

<http://www.juce.com/api/annotated.html>

## Summary

By the end of this chapter you should be familiar with the principles of building user interfaces in JUCE both programmatically and via the IntroJucer application. This chapter has shown you how to create and use JUCE's built-in components, how to construct custom components, and how to perform fundamental drawing operations on-screen. You should read the online documentation for each class introduced during this chapter. You should also examine the code bundle for this book that contains each of the examples developed in this chapter. The code in this bundle also includes more inline comments for each of the examples. The next chapter covers a range of non-GUI classes although many of these will be useful for managing some elements of user interface functionality.

# 3

## Essential Data Structures

JUCE includes a range of important data structures, many of which could be seen as replacements for some of the standard library classes. This chapter introduces the essential classes for JUCE development. In this chapter we will cover the following topics:

- Understanding the numerical types
- Specifying and manipulating strings of text using the `String` class
- Measuring and displaying time
- Specifying file paths in a cross-platform manner using the `File` class (including access to the user's home space, the `Desktop` and `Documents` locations)
- Using dynamically allocated arrays: the `Array` class
- Employing smart pointer classes

By the end of this chapter, you will be able to create and manipulate data in a range of JUCE's essential classes.

## Understanding the numerical types

The word size of some the basic data types (`char`, `int`, `long`, and so on) varies across platforms, compilers, and CPU architectures. A good example is the type `long`. In Xcode on Mac OS X, `long` is 32 bits wide when compiling 32-bit code and 64 bits wide when compiling 64-bit code. In Microsoft Visual Studio on Windows, `long` is always 32 bits wide. (The same applies to the unsigned versions too.) JUCE defines a handful of primitive types to assist the writing of platform-independent code. Many of these have familiar names and may be the same names used in other libraries and frameworks in use by your code. These types are defined in the `juce` namespace; therefore, can be disambiguated using the `juce::` prefix if necessary. These primitive types are: `int8` (8-bit signed integer), `uint8` (8-bit unsigned integer), `int16` (16-bit signed integer), `uint16` (16-bit unsigned integer), `int32` (32-bit signed integer), `uint32` (32-bit unsigned integer), `int64` (64-bit signed integer), `uint64` (64-bit unsigned integer), `pointer_sized_int` (a signed integer that is the same word size as a pointer on the platform), `pointer_sized_uint` (an unsigned integer that is the same word size as a pointer on the platform), and `juce_wchar` (a 32-bit Unicode character type).

In many cases the built-in types are sufficient. For example, JUCE internally makes use of the `int` data type for a number of purposes, but the preceding types are available where the word size is critical. In addition to this, JUCE does not define special data types for `char`, `float`, or `double`. Both floating-point types are assumed to be compliant with IEEE 754, and the `float` data type is assumed to be 32 bits wide and the `double` data type 64 bits wide.

One final utility in this regard addresses the issue that writing 64-bit literals in code differs across compilers. The `literal64bit()` macro can be used to write such literals if needed:

```
int64 big = literal64bit (0x1234567890);
```

JUCE also declares some basic template types for defining certain geometry; the `Component` class uses these in particular. Some useful examples are `Point<ValueType>`, `Line<ValueType>`, and `Rectangle<ValueType>`.

## Specifying and manipulating text strings

In JUCE, text is generally manipulated using the `String` class. In many ways, this class may be seen as an alternative to the C++ Standard Library `std::string` class. We have already used the `String` class for the basic operations in earlier chapters. For example, in *Chapter 2, Building User Interfaces*, strings were used to set the text appearing on a `TextButton` object and used to store a dynamically changing string to display in response to mouse activity. Even though these examples were quite simple, they harnessed the power of the `String` class to make setting and manipulating the strings straightforward for the user.

The first way this is achieved is through storing strings using **reference counted** objects. That is to say, when a string is created, behind the scenes JUCE allocates some memory for the string, stores the string, and returns a `String` object that refers to this allocated memory in the background. Straight copies of this string (that is, without any modifications) are simply new `String` objects that refer to this same shared memory. This helps keep code efficient by allowing `String` objects to be passed by value between functions, without the potential overhead of copying large chunks of memory in the process.

To illustrate some of these features, we will use a console, rather than a Graphical User Interface (GUI), application in the first instance. Create a new Introjucer project named `Chapter03_01`; changing the **Project Type** to **Console Application**, and only selecting **Create a Main.cpp file** in the **Files to Auto-Generate** menu. Save the project and open it into your Integrated Development Environment (IDE).

## Posting log messages to the console

To post messages to the console window, it is best to use JUCE's `Logger` class. Logging can be set to log a text file, but the default behavior is to send the logging messages to the console. A simple "Hello world!" project using a JUCE `String` object and the `Logger` class is shown as follows:

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

int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();
    String message ("Hello world!");
    log->writeToLog (message);

    return 0;
}
```



The first line of code in the `main()` function stores a pointer to the current logger such that we can reuse it a number of times in later examples. The second line creates a JUCE `String` object from the literal C string `"Hello world!"`, and the third line sends this string to the logger using its `writeToLog()` function. Build and run this application, and the console window should look something like the following:

```
JUCE v2.1.2
Hello world!
```

JUCE reports the first line automatically; this may be different if you have a later version of JUCE from the GIT repository. This is followed by any logging messages from your application.

## String manipulation

While this example is more complex than an equivalent using standard C strings, the power of JUCE's `String` class is delivered through the storage and manipulation of strings. For example, to concatenate strings, the `+` operator is overloaded for this purpose:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();
    String hello ("Hello");
    String space (" ");
    String world ("world!");
    String message = hello + space + world;

    log->writeToLog (message);

    return 0;
}
```

Here, separate strings are constructed from literals for `"Hello"`, the space in between, and `"world!"`, then the final message string is constructed by concatenating all three. The stream operator `<<` may also be used for this purpose for a similar result:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();
    String hello ("Hello");
    String space (" ");
    String world ("world!");
    String message;

    message << hello;
```

---

```

    message << space;
    message << world;

    log->writeToLog (message);

    return 0;
}

```

The stream operator concatenates the right-hand side of the expression onto the left-hand side of the expression, in-place. In fact, using this simple case, the << operator is equivalent to the += operator when applied to strings. To illustrate this, replace all the instances of << with += in the code.

The main difference is that the << operator may be more conveniently chained into longer expressions without additional parentheses (due to the difference between the precedence in C++ of the << and += operators). Therefore, the concatenation can be done all on one line, as with the + operator, if needed:

```

int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();
    String message;

    message << "Hello" << " " << "world!";

    log->writeToLog (message);

    return 0;
}

```

To achieve the same results with += would require cumbersome parentheses for each part of the expression: ((message += "Hello") += " ") += "world!").

The way the internal reference counting of strings works in JUCE means that you rarely need to be concerned about unintended side effects. For example, the following listing works as you might expect from reading the code:

```

int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();
    String string1 ("Hello");
    String string2 = string1;

    string1 << " world!";

    log->writeToLog ("string1: " + string1);
    log->writeToLog ("string2: " + string2);

    return 0;
}

```

This produces the following output:

```
string1: Hello world!  
string2: Hello
```

Breaking this down into steps, we can see what happens:

- `String string1 ("Hello");`: The `string1` variable is initialized with a literal string.
- `String string2 = string1;`: The `string2` variable is initialized with `string1`; they now refer to exactly the same data behind the scenes.
- `string1 << " world!";`: The `string1` variable has another literal string appended. At this point `string1` refers to a completely new block of memory containing the concatenated string.
- `log->writeToLog ("string1: " + string1);`: This logs `string1`, showing the concatenated string `Hello world!`.
- `log->writeToLog ("string2: " + string2);`: This logs `string2`; this shows that `string1` still refers to the initial string `Hello`.

One really useful feature of the `String` class is its numerical conversion capabilities. Generally, you can pass a numerical type to a `String` constructor, and the resulting `String` object will represent that numerical value. For example:

```
String intString (1234);    // string will be "1234"  
String floatString (1.25f); // string will be "1.25"  
String doubleString (2.5); // string will be "2.5"
```

Other useful features are conversions to uppercase and lowercase. Strings may also be compared using the `==` operator.

## Measuring and displaying time

The `JUCE Time` class provides a cross-platform way to specify, measure, and format date and time information in a human-readable fashion. Internally, the `Time` class stores a value in milliseconds relative to midnight on 1st January 1970. To create a `Time` object that represents the current time, use `Time::getCurrentTime()` like the following:

```
Time now = Time::getCurrentTime();
```

To bypass the creation of the `Time` object, you can access the millisecond counter as a 64-bit value directly:

```
int64 now = Time::currentTimeMillis();
```

The `Time` class also provides access to a 32-bit millisecond counter that measures time since system startup:

```
uint32 now = Time::getMillisecondCounter();
```

The important point to note about `Time::getMillisecondCounter()` is that it is independent of the system time, and would be unaffected by changes to the system time either by the user changing the time, changes due to national daylight saving, and so on.

## Displaying and formatting time information

Displaying time information is straightforward; the following example gets the current time from the operating system, formats it as a string, and sends it to the console output:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    Time time (Time::getCurrentTime());

    bool includeDate = true;
    bool includeTime = true;
    bool includeSeconds = true;
    bool use24HourClock = true;

    String timeStr (time.toString (includeDate, includeTime,
                                   includeSeconds, use24HourClock));

    log->writeToLog ("the time is: " + timeStr);

    return 0;
}
```

This illustrates the four option flags available to the `Time::toString()` function. The output on the console will be something like:

```
the time is: 7 Jul 2013 15:05:55
```

For more comprehensive options, the `Time::formatted()` function allows the user to specify a format using a special format string (using a system equivalent to the standard C `strftime()` function). Alternatively, you can obtain the various parts of the date and time information (day, month, hour, minute, time zone, and so on), and combine them into a string yourself. For example, the same preceding format can be achieved as follows:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    Time time (Time::getCurrentTime());

    String timeStr;

    bool threeLetterMonthName = true;

    timeStr << time.getDayOfMonth() << " ";
    timeStr << time.getMonthName (threeLetterMonthName) << " ";
    timeStr << time.getYear() << " ";
    timeStr << time.getHours() << ":";
    timeStr << time.getMinutes() << ":";
    timeStr << time.getSeconds();

    log->writeToLog ("the time is: " + timeStr);

    return 0;
}
```

## Manipulating time data

Time objects may also be manipulated (with the help from the `RelativeTime` class) and compared with other Time objects. The following example shows the creation of three time values, based on the current time, using a one-hour offset:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    Time time (Time::getCurrentTime());
    RelativeTime oneHour (RelativeTime::hours (1));
```

---

```

    Time oneHourAgo (time - oneHour);
    Time inOneHour (time + oneHour);
    Time inTwoHours (inOneHour + oneHour);

    log->writeToLog ("the time is:" +
                    time.toString (true, true, true, true));
    log->writeToLog ("one hour ago was:" +
                    oneHourAgo.toString (true, true, true, true));
    log->writeToLog ("in one hour it will be:" +
                    inOneHour.toString (true, true, true, true));
    log->writeToLog ("in two hours it will be:" +
                    inTwoHours.toString (true, true, true, true));

    return 0;
}

```

The output of this should be something like this:

```

the time is:          7 Jul 2013 15:42:27
one hour ago was:     7 Jul 2013 14:42:27
in one hour it will be: 7 Jul 2013 16:42:27
in two hours it will be: 7 Jul 2013 17:42:27

```

To compare two `Time` objects, the standard comparison operators may be used. For example, you could wait for a specific time, like the following:

```

int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    Time now (Time::getCurrentTime());
    Time trigger (now + RelativeTime (5.0));

    log->writeToLog ("the time is now:      " +
                    now.toString (true, true, true, true));

    while (Time::getCurrentTime() < trigger) {
        Thread::sleep (10);
        log->writeToLog ("waiting...");
    }

    log->writeToLog ("the time has reached: " +
                    trigger.toString (true, true, true, true));

    return 0;
}

```

Two things to note here are that:

- The value passed to the `RelativeTime` constructor is in seconds (all the other time values need to use one of the static functions as shown earlier for hours, minutes, and so on).
- The call to `Thread::sleep()` uses values in milliseconds and this sleeps the calling thread. The `Thread` class will be examined further in *Chapter 5, Helpful Utilities*.

## Measuring time

The time values returned from the `Time::getCurrentTime()` function should be accurate for most purposes, but as pointed out earlier, the *current time* could be changed by the user modifying the system time. An equivalent to the preceding example, using `Time::getMillisecondCounter()` that is not susceptible to such changes, is shown as follows:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    uint32 now = Time::getMillisecondCounter();
    uint32 trigger = now + 5000;

    log->writeToLog ("the time is now: " +
                    String (now) + "ms");

    while (Time::getMillisecondCounter() < trigger) {
        Thread::sleep (10);
        log->writeToLog ("waiting...");
    }

    log->writeToLog ("the time has reached: " +
                    String (trigger) + "ms");

    return 0;
}
```

Both the `Time::getCurrentTime()` and `Time::getMillisecondCounter()` functions have a similar accuracy, which is within a few milliseconds on most platforms. However, the `Time` class also provides access to a higher resolution counter that returns values as a double precision (64-bit) floating-point value. This function is `Time::getMillisecondCounterHiRes()`, and is also relative to the system start-up as is the value returned from the `Time::getMillisecondCounter()` function. One application of this is to measure the time that certain pieces of code have taken to execute, as shown in the following example:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    double start = Time::getMillisecondCounterHiRes();

    log->writeToLog ("the time is now: " +
                    String (start) + "ms");

    float value = 0.f;
    const int N = 10000;

    for (int i = 0; i < N; ++i)
        value += 0.1f;

    double duration = Time::getMillisecondCounterHiRes() - start;

    log->writeToLog ("the time taken to perform " + String (N) +
                    " additions was: " + String (duration) + "ms");

    return 0;
}
```

This records the current time by polling the higher resolution counter, performing a large number of floating point additions, and polling the higher resolution counter again to determine the duration between these two points in time. The output should be something like this:

```
the time is now: 267150354ms
the time taken to perform 10000 additions was: 0.0649539828ms
```

Of course, the results here are dependent on the optimization settings in the compiler and the runtime system.



## Specifying file paths

JUCE provides a relatively cross-platform way of specifying and manipulating file paths using the `File` class. In particular, this provides a means of accessing various special directories on the user's system, such as the `Desktop` directory, their user `Documents` directory, application preferences directories, and so on. The `File` class also provides functions for accessing information about a file (for example, creation date, modification date, file size) and basic mechanisms for reading and writing file contents (although other techniques may be more appropriate for large or complex files). In the following example, a string is written to a text file on disk (using the `File::replaceWithText()` function), then read back into a second string (using the `File::loadFileAsString()` function), and displayed in the console:

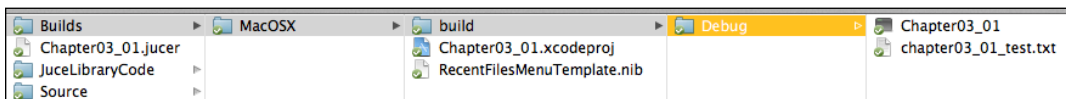
```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    String text ("The quick brown fox jumps over the lazy dog.");
    File file ("./chapter03_01_test.txt");
    file.replaceWithText (text);
    String fileText = file.loadFileAsString();

    log->writeToLog ("fileText: " + fileText);

    return 0;
}
```

The `File` object in this case is initialized with the path `./chapter03_01_test.txt`. It should be noted that this file may not exist at this point, and on first run it will not exist until the call to the `File::replaceWithText()` function (and on subsequent runs this file will exist, but will be overwritten at that point). The `./` character sequence at the front of this path is a common idiom specifying that the remainder of the path should be relative to the current directory (or current working directory). In this simple case, the current working directory is likely to be the directory where the executable file is located. The following screenshot shows this location relative to the Introjucer project on the Mac platform:



This is not a reliable method; however, it will work if the working directory is specifically where you want to save a file.

## Accessing various special directory locations

It is more precise to use one of the `File` class's special locations, as shown as follows:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    String text ("The quick brown fox jumps over the lazy dog.");
    File exe (File::getSpecialLocation(
        File::currentExecutableFile));
    File exeDir (exe.getParentDirectory());
    File file (exeDir.getChildFile ("chapter03_01_test.txt"));
    file.replaceWithText (text);
    String fileText = file.loadFileAsString();

    log->writeToLog ("fileText: " + fileText);

    return 0;
}
```

The steps for accessing the file location in this directory are split across several lines for clarity in this code. Here, you can see the code to obtain the location of the current executable file, then its parent directory, and then create a file reference for our text file that is relative to this directory. Much of this code may be compacted on a single logical line using a chain of function calls:

```
...
File file (File::getSpecialLocation(
    File::currentExecutableFile)
    .getParentDirectory()
    .getChildFile ("chapter03_01_test.txt"));
...
```

Due to the length of some of the identifiers in this code and the page width in this book, this code still occupies four physical lines of code. Nevertheless, this illustrates how you can employ this function calls to suit your needs and preferences for code layout.

## Obtaining various information about files

The `File` class can provide useful information about files. One important test is whether a file exists; this can be determined using `File::exists()`. If a file does exist, then more information may be obtained, such as its creation date, modification date, and size. These are illustrated in the following example:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    File file (File::getSpecialLocation(File::currentExecutableFile)
               .getParentDirectory()
               .getChildFile("chapter03_01_test.txt"));

    bool fileExists = file.exists();

    if (!fileExists) {
        log->writeToLog ("file " +
                        file.getFileName() +
                        " does not exist");

        return -1;
    }

    Time creationTime = file.getCreationTime();
    Time modTime = file.getLastModificationTime();
    int64 size = file.getSize();

    log->writeToLog ("file " +
                    file.getFileName() + " info:");
    log->writeToLog ("created: " +
                    creationTime.toString(true, true, true, true));
    log->writeToLog ("modified:" +
                    modTime.toString(true, true, true, true));
    log->writeToLog ("size:" +
                    String(size) + " bytes");

    return 0;
}
```

Assuming you ran all of the preceding examples, the file should exist on your system and the information will be reported in the console something like as follows:

```
file chapter03_01_test.txt info:
created: 8 Jul 2013 17:08:25
modified: 8 Jul 2013 17:08:25
size: 44 bytes
```

## Other special locations

In addition to `File::currentExecutableFile`, other special locations known to JUCE are:

- `File::userHomeDirectory`
- `File::userDocumentsDirectory`
- `File::userDesktopDirectory`
- `File::userApplicationDataDirectory`
- `File::commonApplicationDataDirectory`
- `File::tempDirectory`
- `File::currentExecutableFile`
- `File::currentApplicationFile`
- `File::invokedExecutableFile`
- `File::hostApplicationPath`
- `File::globalApplicationsDirectory`
- `File::userMusicDirectory`
- `File::userMoviesDirectory`
- `File::userPicturesDirectory`

Each of these names is fairly self-explanatory. In some cases, these special locations are not applicable on some platforms. For example, there is no such thing as the Desktop on the iOS platform.

## Navigating directory structures

Ultimately, a `File` object resolves to an absolute path on the user's system. This can be obtained using the `File::getFullPathName()` function if needed:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    File file (File::getSpecialLocation(
        File::currentExecutableFile)
        .getParentDirectory()
        .getChildFile ("chapter03_01_test.txt"));
    log->writeToLog ("file path: " + file.getFullPathName());

    return 0;
}
```

In addition to this, the relative path passed to `File::getChildFile()` can contain one or more references to parent directories using the double period notation (that is, the `".."` character sequence). In this next example, we create a simple directory structure as shown in the screenshot following this code listing:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    File root (File::getSpecialLocation (File::userDesktopDirectory)
        .getChildFile ("Chapter03_01_tests"));
    File dir1 (root.getChildFile ("1"));
    File dir2 (root.getChildFile ("2"));
    File dir1a (dir1.getChildFile ("a"));
    File dir2b (dir2.getChildFile ("b"));

    Result result (Result::ok());

    result = dir1a.createDirectory();

    if (!result.wasOk()) {
        log->writeToLog ("Creating dir 1/a failed");
        return -1;
    }

    result = dir2b.createDirectory();
```

```

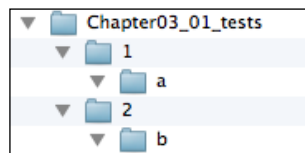
    if (!result.wasOk()) {
        log->writeToLog ("Creating dir 2/b failed");
        return -1;
    }

    File rel = dir1a.getChildFile ("../2/b");

    log->writeToLog ("root: " + root.getFullPathName());
    log->writeToLog ("dir1: " + dir1.getRelativePathFrom (root));
    log->writeToLog ("dir2: " + dir2.getRelativePathFrom (root));
    log->writeToLog ("dir1a: " + dir1a.getRelativePathFrom (root));
    log->writeToLog ("dir2b: " + dir2b.getRelativePathFrom (root));
    log->writeToLog ("rel: " + rel.getRelativePathFrom (root));

    return 0;
}

```



This creates five directories in total, using only two calls to the `File::createDirectory()` function. Since this is dependent on the user's permissions to create files in this directory, the function returns a `Result` object. This contains a state to indicate if the function succeeded or not (which we check with the `Result::wasOk()` function), and more information can be gained about any errors if needed. Each call to the `File::createDirectory()` function ensures that it creates any intermediate directories if required. Therefore, on the first call, it creates the root directory, directory 1, and directory 1/a. On the second call, the root already exists, so it needs only to create directories 2 and 2/a.

The console output for this should be something like this:

```

root: /Users/martinrobinson/Desktop/Chapter03_01_tests
dir1: 1
dir2: 2
dir1a: 1/a
dir2b: 2/b
rel: 2/b

```

Of course, the first line will be different, depending on your system, but the remaining five lines should be the same. These paths are displayed relative to the root of the directory structure we have created using the `File::getRelativePathFrom()` function. Notice that the final line shows that the `rel` object refers to the same directory as the `dir2b` object, but we created this `rel` object relative to the `dir1a` object by using the function call `dir1a.getChildFile("../..2/b")`. That is, we navigate two levels up the directory structure then access the directories below.

The `File` class also includes features to check for a file's existence, to move and copy files within the filesystem (including moving the file to the **Trash** or **Recycle Bin**), and to create legal filenames on particular platforms (for example, avoiding colon and slash characters).

## Using dynamically allocated arrays

While most instances of JUCE objects can be stored in regular C++ arrays, JUCE offers a handful of arrays that are more powerful, somewhat comparable to the C++ Standard Library classes, such as `std::vector`. The JUCE `Array` class offers many features; these arrays can be:

- Dynamically sized; items can be added, removed, and inserted at any index
- Sorted using custom comparators
- Searched for particular content

The `Array` class is a template class; its main template argument, `ElementType`, must meet certain criteria. The `Array` class moves its contents around by copying memory during resizing and inserting elements, this could cause problems with certain kinds of objects. The class passed as the `ElementType` template argument must also have both a copy constructor and an assignment operator. The `Array` class, in particular, works well with primitive types and some commonly used JUCE classes, for example, the `File` and `Time` classes. In the following example, we create an array of integers, add five items to it, and iterate over the array, sending the contents to the console:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    Array<int> array;

    for (int i = 0; i < 5; ++i)
        array.add (i * 1000);
```

---

```
    for (int i = 0; i < array.size(); ++i) {
        int value = array[i];
        log->writeToLog ("array[" + String (i) + "] = " + String (value));
    }

    return 0;
}
```

This should produce the output:

```
array[0]= 0
array[1]= 1000
array[2]= 2000
array[3]= 3000
array[4]= 4000
```

Notice that the JUCE Array class supports the C++ indexing subscript operator []. This will always return a valid value even if the array index is out of bounds (unlike a built-in array). There is a small overhead involved in making this check; therefore, you can avoid the bounds checking by using the `Array::getUnchecked()` function, but you must be certain that the index is within bounds, otherwise your application may crash. The second `for()` loop can be rewritten as follows to use this alternative function, because we have already checked that our indices will be in-range:

```
...
    for (int i = 0; i < array.size(); ++i) {
        int value = array.getUnchecked (i);
        log->writeToLog("array[" + String (i) + "] = " +
                        String (value));
    }
...
```



## Finding the files in a directory

The JUCE library uses the `Array` objects for many purposes. For example, the `File` class can fill an array of `File` objects with a list of child files and directories it contains using the `File::findChildFiles()` function. The following example should post a list of files and directories in your user `Documents` directory to the console:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    File file =
        File::getSpecialLocation (File::userDocumentsDirectory);

    Array<File> childFiles;

    bool searchRecursively = false;
    file.findChildFiles (childFiles,
                        File::findFilesAndDirectories,
                        searchRecursively);

    for (int i = 0; i < childFiles.size(); ++i)
        log->writeToLog (childFiles[i].getFullPathName());

    return 0;
}
```

Here, the `File::findChildFiles()` function is passed the array of `File` objects, to which it should add the result of the search. It is also told to find both files and directories using the value `File::findFilesAndDirectories` (other options are the `File::findDirectories` and `File::findFiles` values). Finally, it is told not to search recursively.

## Tokenizing strings

Although it is possible to use `Array<String>` to hold an array of JUCE `String` objects, there is a dedicated `StringArray` class to offers additional functionality when applying array operations to string data. For example, a string can be **tokenized** (that is, broken up into smaller strings based on whitespace in the original string) using the `String::addTokens()` function, or divided into strings representing lines of text (based on newline character sequences found within the original string) using the `String::addLines()` function. The following example tokenizes a string, then iterates over the resulting `StringArray` object, posting its contents to the console:

```
int main (int argc, char* argv[])
{
    Logger *log = Logger::getCurrentLogger();

    StringArray strings;
    bool preserveQuoted = true;
    strings.addTokens("one two three four five six",
                     preserveQuoted);

    for (int i = 0; i < strings.size(); ++i) {
        log->writeToLog ("strings[" + String (i) + "]=\" +
                        strings[i]);
    }

    return 0;
}
```

## Arrays of components

User interfaces comprising banks of similar controls, such as buttons and sliders, can be managed effectively using arrays. However, the JUCE `Component` class and its subclasses do not meet the criteria for storage as an object (that is, by value) in a JUCE `Array` object. These must be stored as arrays of pointers to these objects instead. To illustrate this, we need a new Introjucer project with a basic window as used throughout *Chapter 2, Building User Interfaces*. Create a new Introjucer project, such as this, name it `Chapter03_02`, and open it into your IDE. To the end of the `MainWindow` constructor in `Main.cpp`, add the following line:

```
setResizable (true, true);
```

In the `MainComponent.h` file change the code to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

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

    void resized();

private:
    Array<TextButton*> buttons;
};

#endif
```

Notice that the `Array` object here is an array of pointers to `TextButton` objects (that is, `TextButton*`). In the `MainComponent.cpp` file change the code to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    for (int i = 0; i < 10; ++i)
    {
        String buttonName;
        buttonName << "Button " << String(i);
        TextButton* button = new TextButton(buttonName);
        buttons.add(button);
        addAndMakeVisible(button);
    }

    setSize(500, 400);
}

MainContentComponent::~MainContentComponent()
{
}
```

---

```

void MainContentComponent::resized()
{
    Rectangle<int> rect (10, 10, getWidth() - 20, getHeight() - 20);

    int buttonHeight = rect.getHeight() / buttons.size();

    for (int i = 0; i < buttons.size(); ++i) {
        buttons[i]->setBounds (rect.getX(),
                               i * buttonHeight + rect.getY(),
                               rect.getWidth(),
                               buttonHeight);
    }
}

```

Here, we create 10 buttons and using a `for()` loop, adding these buttons to an array, and basing the name of the button on the loop counter. The buttons are allocated using the `new` operator (rather than the static allocation used in *Chapter 2, Building User Interfaces*), and it is these pointers that are stored in the array. (Notice also, that there is no need for the `&` operator in the function call to `Component::addAndMakeVisible()` because the value is already a pointer.) In the `resized()` function, we use a `Rectangle<int>` object to create a rectangle that is inset from the `MainContentComponent` object's bounds rectangle by 10 pixels all the way around. The buttons are positioned within this smaller rectangle. The height for each button is calculated by dividing the height of our rectangle by the number of buttons in the button array. The `for()` loop then positions each button, based on its index within the array. Build and run the application; its window should present 10 buttons arranged in a single column.

There is one major flaw with the preceding code. The buttons allocated with the `new` operator are never deleted. The code should run fine, although you will get an assertion failure when the application is exited. The message into the console will be something like:

```

*** Leaked objects detected: 10 instance(s) of class TextButton
JUICE Assertion failure in juce_LeakedObjectDetector.h:95

```

To solve this, we could delete the buttons in the `MainComponent` destructor like so:

```

MainContentComponent::~MainContentComponent()
{
    for (int i = 0; i < buttons.size(); ++i)
        delete buttons[i];
}

```

However, it is very easy to forget to do this kind of operation when writing complex code.

## Using the OwnedArray class

JUCE provides a useful alternative to the Array class that is dedicated to pointer types: the OwnedArray class. The OwnedArray class always stores pointers, therefore should not include the \* character in the template parameter. Once a pointer is added to an OwnedArray object, it takes ownership of the pointer and will take care of deleting it when necessary (for example, when the OwnedArray object itself is destroyed). Change the declaration in the MainComponent.h file, as highlighted in the following:

```
...
private:
    OwnedArray<TextButton> buttons;
};
```

You should also remove the code from the destructor in the MainComponent.cpp file, because deleting objects more than once is equally problematic:

```
...
MainContentComponent::~MainContentComponent()
{
}
...
```

Build and run the application, noticing that the application will now exit without problems.

This technique can be extended to using broadcasters and listeners. Create a new GUI-based Introjucer project as before, and name it Chapter03\_03. Change the MainComponent.h file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

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

    void resized();
    void buttonClicked (Button* button);

private:
    OwnedArray<Button> buttons;
    Label label;
};

#endif
```

This time we use an `OwnedArray<Button>` object rather than an `OwnedArray<TextButton>` object. This simply avoids the need to typecast our button pointers to different types when searching for the pointers in the array, as we do in the following code. Also, notice here that we added a `Label` object to our component, made our component a button listener, and that we do not need a destructor. Change the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    for (int i = 0; i < 10; ++i) {
        String buttonName;
        buttonName << "Button " << String(i);
        TextButton* button = new TextButton(buttonName);
        button->addListener(this);
        buttons.add(button);
        addAndMakeVisible(button);
    }

    addAndMakeVisible(&label);
    label.setJustificationType(Justification::centred);
    label.setText("no buttons clicked", dontSendNotification);

    setSize(500, 400);
}

void MainContentComponent::resized()
{
    Rectangle<int> rect(10, 10,
                      getWidth() / 2 - 20, getHeight() - 20);

    int buttonHeight = rect.getHeight() / buttons.size();

    for (int i = 0; i < buttons.size(); ++i) {
        buttons[i]->setBounds(rect.getX(),
                              i * buttonHeight + rect.getY(),
                              rect.getWidth(),
                              buttonHeight);
    }
}
```

```
        label.setBounds (rect.getRight(),
                        rect.getY(),
                        getWidth() - rect.getWidth() - 10,
                        20);
    }

    void MainContentComponent::buttonClicked (Button* button)
    {
        String labelText;
        int buttonIndex = buttons.indexOf (button);
        labelText << "Button clicked: " << String (buttonIndex);
        label.setText (labelText, dontSendNotification);
    }
```

Here, we add the label in the constructor, reduce the width of the bank of buttons to occupy only the left half of the component, and position the label at the top in the right-half. In the button listener callback, we can obtain the index of the button using the `OwnedArray::indexOf()` function to search for the pointer (incidentally, the `Array` class also has an `indexOf()` function for searching the items). Build and run the application and notice that our label reports which button was clicked. Of course, the elegant thing about this code is that we need only change the value in the `for()` loop when the buttons are created in our constructor to change the number of buttons that are created; everything else works automatically.

## Other banks of controls

This approach may be applied to other banks of controls. The following example creates a bank of sliders and labels, keeping each corresponding component updated with the appropriate value. Create a new GUI-based Introjucer project, and name it `Chapter03_04`. Change the `MainComponent.h` file to:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component,
                             public Slider::Listener,
                             public Label::Listener
{
public:
    MainContentComponent();
```

---

```

    void resized();
    void sliderValueChanged (Slider* slider);
    void labelTextChanged (Label* label);

private:
    OwnedArray<Slider> sliders;
    OwnedArray<Label> labels;
};

#endif

```

Here, we have arrays of sliders and labels and our component is both a label listener and a slider listener. Now, update the `MainComponent.cpp` file to contain the include directive, the constructor, and the `resized()` function:

```

#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    for (int i = 0; i < 10; ++i) {
        String indexString (i);
        String sliderName ("slider" + indexString);
        Slider* slider = new Slider (sliderName);
        slider->setTextBoxStyle (Slider::NoTextBox, false, 0, 0);
        slider->addListener (this);
        sliders.add (slider);
        addAndMakeVisible (slider);

        String labelName ("label" + indexString);
        Label* label = new Label (labelName,
                                String (slider->getValue()));
        label->setEditable (true);
        label->addListener (this);
        labels.add (label);
        addAndMakeVisible (label);
    }

    setSize (500, 400);
}

void MainContentComponent::resized()
{

```



```
Rectangle<int> slidersRect (10, 10,
                           getWidth() / 2 - 20,
                           getHeight() - 20);
Rectangle<int> labelsRect (slidersRect.getRight(), 10,
                           getWidth() / 2 - 20,
                           getHeight() - 20);

int cellHeight = slidersRect.getHeight() / sliders.size();

for (int i = 0; i < sliders.size(); ++i) {
    sliders[i]->setBounds (slidersRect.getX(),
                           i * cellHeight + slidersRect.getY(),
                           slidersRect.getWidth(),
                           cellHeight);
    labels[i]->setBounds (labelsRect.getX(),
                           i * cellHeight + labelsRect.getY(),
                           labelsRect.getWidth(),
                           cellHeight);
}
```

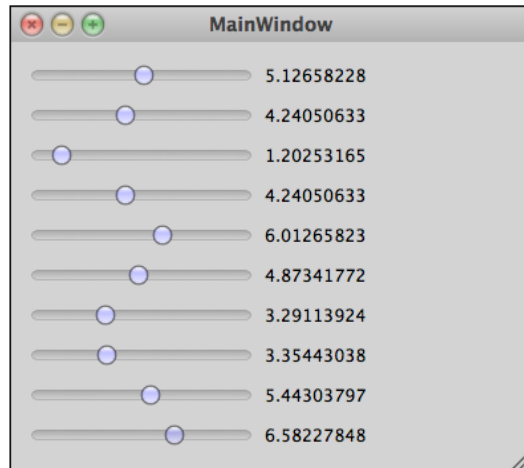
Here, we use a `for()` loop to create the components and add them to the corresponding arrays. In the `resized()` function, we create two helper rectangles, one for the bank of sliders and one for the bank of labels. These are positioned to occupy the left half and right half of the main component respectively.

In the listener callback functions, the index of the broadcasting component is looked up in its array, and this index is used to set the value of the other corresponding component. Add these listener callback functions to the `MainComponent.cpp` file:

```
void MainContentComponent::sliderValueChanged (Slider* slider)
{
    int index = sliders.indexOf (slider);
    labels[index]->setText (String (slider->getValue()),
                           sendNotification);
}

void MainContentComponent::labelTextChanged (Label* label)
{
    int index = labels.indexOf (label);
    sliders[index]->setValue (label->getText().getDoubleValue());
}
```

Here, we use the `String` class to perform the numerical conversions. After moving some of the sliders, the application window should look similar to the following screenshot:



Hopefully, these examples illustrate the power of combining JUCE array classes with other JUCE classes to write elegant, readable, and powerful code.

## 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.

## Summary

This chapter has outlined some of the core classes in JUCE that provide a foundation for building JUCE applications and provide a framework for building applications that are idiomatic to the JUCE style. These classes provide further foundations for the remainder of this book. Each of these classes contains far more functionality than outlined here. Again, it is essential that you review the JUCE class documentation for each of the classes introduced in this chapter. Many of these classes are used heavily in the JUCE Demo application and the code for the Introjucer application. These should also serve as useful for further reading. The next chapter introduces classes for handling files, especially media files, such as image and sound files.

# 4

## Using Media Files

JUCE provides its own classes for reading and writing files and many helper classes for specific media formats. This chapter introduces the main examples of these classes. In this chapter we will cover the following topics:

- Using simple input and output streams
- Reading and writing image files
- Playing audio files
- Working with the Binary Builder tool to turn binary files into source code

By the end of this chapter, you will be able to manipulate a range of media files using JUCE.

### Using simple input and output streams

In *Chapter 3, Essential Data Structures*, we introduced the JUCE `File` class, which is used for specifying file paths in a cross-platform manner. In addition, the `File` class includes some convenience functions for reading and writing files as blocks of data or strings of text. In many cases these functions are sufficient, but in others, raw access to input and output streams may be more useful.

## Reading and writing text files

First, create a console application project in the Introjucer application and name it `Chapter04_01`. In this simple example, we will declare two functions, one for writing text to the file—`writeFile()`, and one for reading the contents of the file—`readFile()`. Each of these functions is passed the same file path reference created in the way we did in *Chapter 3, Essential Data Structures*. Replace the contents of the `Main.cpp` file with the following, where we declare the file reading and writing functions, and define a `main()` function:

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

void writeFile (File const& file);
void readFile (File const& file);

int main (int argc, char* argv[])
{
    File file (File::getSpecialLocation(File::currentExecutableFile)
               .getParentDirectory()
               .getChildFile ("chapter04_01_test.txt"));

    writeFile (file);
    readFile (file);

    return 0;
}
```

Then, add the definition for the `writeFile()` function:

```
void writeFile (File const& file)
{
    Logger *log = Logger::getCurrentLogger();
    FileOutputStream stream (file);

    if (!stream.openedOk()) {
        log->writeToLog ("failed to open stream");
        return;
    }

    stream.setPosition (0);
    stream.truncate();

    String text ("The quick brown fox jumps over the lazy dog.");

    bool asUTF16 = false;
    bool byteOrderMark = false;
    stream.writeText (text, asUTF16, byteOrderMark);
}
```

Here, we create a `FileOutputStream` object, passing it the `File` object that refers to the file path. The `FileOutputStream` class inherits from the base class `OutputStream` that represents the general notions of writing data to a stream. There can be other types of output stream, such as the `MemoryOutputStream` class for writing data to areas of computer memory in a stream-like manner. The default behavior of the `FileOutputStream` class on construction is to position the stream's write position at the end of the file if the file already exists (or to create an empty file if it doesn't). The calls to the `FileOutputStream::setPosition()` and `FileOutputStream::truncate()` functions effectively empty the file each time before we write it. Of course, in a real application you may not want to do this each time. The call to the `FileOutputStream::writeText()` function is almost equivalent to the `File::appendText()` function, although the flags for controlling the output in Unicode UTF16 format are implicit for the `File::appendText()` function, but need to be specified explicitly for the `FileOutputStream::writeText()` function. Here, we write the data in UTF8 format by setting both flags to `false`.



The UTF8 format is probably most convenient, because the text we are writing is plain ASCII text, which is compatible with UTF8 encoding.



Finally, add the definition for the `readFile()` function:

```
void readFile (File const& file)
{
    Logger *log = Logger::getCurrentLogger();
    FileInputStream stream (file);

    if (!stream.openedOk()) {
        log->writeToLog ("failed to open stream");
        return;
    }

    log->writeToLog ("fileText: " +
        stream.readEntireStreamAsString());
}
```

Here, we attempt to read the entire stream into a `String`, and post it to the log. We use a `FileInputStream` object, which inherits from the more general `InputStream` class. In both the `writeFile()` and `readFile()` functions we check that the streams opened successfully before proceeding. In addition to this, the stream objects gracefully close the streams when they go out of scope.



## Reading and writing binary files

The output and input streams can be used for binary data too, and offer much greater functionality over the `File` class convenience functions. Here, you can write raw numerical data, and choose the byte order for multibyte data types.

Create a new console application in the Introjucer application and name it `Chapter04_02`. The following example writes `int`, `float`, and `double` data types to a file, and then reads this data back in, posting the result to the log. Replace the contents of `Main.cpp` file with the following code:

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

void writeFile (File const& file);
void readFile (File const& file);

int main (int argc, char* argv[])
{
    File file (File::getSpecialLocation(File::currentExecutableFile)
               .getParentDirectory()
               .getChildFile ("chapter04_02_test.bin"));

    writeFile (file);
    readFile (file);

    return 0;
}

void writeFile (File const& file)
{
    Logger *log = Logger::getCurrentLogger();
    FileOutputStream stream (file);

    if (!stream.openedOk()) {
        log->writeToLog ("failed to open stream");
        return;
    }
}
```

```
stream.setPosition (0);
stream.truncate();
stream.writeInt (1234);
stream.writeFloat (3.142);
stream.writeDouble (0.000000001);
}

void readFile (File const& file)
{
    Logger *log = Logger::getCurrentLogger();
    FileInputStream stream (file);

    if (!stream.openedOk()) {
        log->writeToLog ("failed to open stream");
        return;
    }

    log->writeToLog("readInt: " + String (stream.readInt()));
    log->writeToLog("readFloat: " + String (stream.readFloat()));
    log->writeToLog("readDouble: " + String (stream.readDouble()));
}
```

The `OutputStream` and `InputStream` classes and their respective subclasses, support writing and reading the various built-in types using functions `writeInt()`, `writeFloat()`, `readInt()`, `readFloat()`, and so on. These versions of the functions write these multi-byte types using little endian byte order. For file formats requiring big endian storage, there are equivalent functions `writeIntBigEndian()`, `writeFloatBigEndian()`, `readIntBigEndian()`, `readFloatBigEndian()`, and so on.

The JUCE stream classes are useful but quite low level. For many purposes, JUCE already includes high-level classes for reading and writing specific file types. Of course, these are built on top of the stream classes, but, unless you are dealing with a custom data format, it is likely to be more sensible to use built-in functionality for handling things such as images, audio, and other formats such as **Extensible Markup Language (XML)** and **JavaScript Object Notation (JSON)**.

## Reading and writing image files

JUCE includes built-in support for reading and writing GIF, PNG, and JPEG image files. JUCE also includes its own `Image` class for holding bitmap images. The following example illustrates how to present a native file browser to choose an image file, load the image file, and display it in an `ImageComponent` object. Create a new GUI project in the IntroJucer application with a basic window named `Chapter04_03`, and make the window resizable in the `Main.cpp` file, as we did in earlier chapters. You should then change the `MainComponent.h` file to contain:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component,
                             public Button::Listener
{
public:
    MainContentComponent();
    void resized();
    void buttonClicked (Button* button);

private:
    TextButton readImageButton;
    ImageComponent imageComponent;
    Image image;
};
#endif
```

Change `MainComponent.cpp` to contain:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: readImageButton ("Read Image File...")
{
    addAndMakeVisible (&readImageButton);
    addAndMakeVisible (&imageComponent);

    readImageButton.addListener (this);
```

```
    setSize (500, 400);
}

void MainContentComponent::resized()
{
    int buttonHeight = 20;
    int margin = 10;
    readFileButton.setBounds(margin, margin,
                             getWidth() - margin * 2, buttonHeight);
    imageComponent.setBounds(margin, margin + buttonHeight + margin,
                             getWidth() - margin * 2,
                             getHeight() - buttonHeight - margin * 3);
}

void MainContentComponent::buttonClicked (Button* button)
{
    if (&readFileButton == button)
    {
        FileChooser chooser ("Choose an image file to display...");

        if (chooser.browseForFileToOpen()) {
            image = ImageFileFormat::loadFrom (chooser.getResult());

            if (image.isValid())
                imageComponent.setImage (image);
        }
    }
}
```

Here, we create a `FileChooser` object in response to the user clicking on the **Read Image File...** button. This presents a native dialog window that allows the user to choose a file. We use the `ImageFileFormat::loadFrom()` function to attempt to load the file as an image. Because we didn't limit the file types displayed or enabled in the file chooser, the user may not have chosen a valid image file. We check the validity of the image, and if it is valid we pass the loaded image to the `ImageComponent` object for display. The `ImageComponent` class has various options to control the way the image is positioned and scaled, depending on how the original image size and component rectangle compare. These can be controlled using the `ImageComponent::setImagePlacement()` function. The following screenshot shows the application that reads an image file:



The `Image` class is similar to the `String` class, in that it uses a reference-counted object internally such that several `Image` objects may share the same internal data.

## Manipulating image data

In the next example we will add a slider to control the brightness of the displayed image and a button to write this processed image as a PNG file. Change the contents of the `MainComponent.h` file, where the changes are highlighted in the following code listing:

```
#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:
    TextButton readImageButton;
    ImageComponent imageComponent;
    Slider brightnessSlider;
    TextButton writeImageButton;
    Image origImage, procImage;
};
#endif
```

Now replace the `MainComponent.cpp` file with the include directive and the constructor:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: readFileButton ("Read Image File..."),
  writeFileButton ("Write Image File...")
{
    brightnessSlider.setRange (0.0, 10.0);
    addAndMakeVisible (&readFileButton);
    addAndMakeVisible (&imageComponent);
    addAndMakeVisible (&brightnessSlider);
    addAndMakeVisible (&writeFileButton);

    readFileButton.addListener (this);
    writeFileButton.addListener (this);
    brightnessSlider.addListener (this);

    setSize (500, 400);
}
```

Add the `resized()` function that positions the components:

```
void MainContentComponent::resized()
{
    int controlHeight = 20;
    int margin = 10;
    int width = getWidth() - margin * 2;

    readFileButton.setBounds
        (margin, margin, width, controlHeight);
    imageComponent.setBounds
        (margin, readFileButton.getBottom() + margin, width,
         getHeight() - (controlHeight + margin) * 3 - margin * 2);
    brightnessSlider.setBounds
        (margin, imageComponent.getBottom() + margin,
         width, controlHeight);
    writeFileButton.setBounds
        (margin, brightnessSlider.getBottom() + margin,
         width, controlHeight);
}
```

Add the `buttonClicked()` function that responds to the button interactions:

```
void MainContentComponent::buttonClicked (Button* button)
{
    if (&readFileButton == button) {
        FileChooser chooser ("Choose an image file to display...");

        if (chooser.browseForFileToOpen()) {
            origImage = ImageFileFormat::loadFrom (chooser.getResult());

            if (origImage.isValid()) {
                procImage = origImage.createCopy();
                imageComponent.setImage (procImage);
            }
        }
    } else if (&writeFileButton == button) {
        if (procImage.isValid()) {
            FileChooser chooser ("Write processed image to file...");

            if (chooser.browseForFileToSave (true)) {
                FileOutputStream stream (chooser.getResult());
                PNGImageFormat pngImageFormat;
                pngImageFormat.writeImageToStream (procImage, stream);
            }
        }
    }
}
```

Finally, add the `sliderValueChanged()` function that responds to the slider interaction:

```
void MainContentComponent::sliderValueChanged (Slider* slider)
{
    if (&brightnessSlider == slider) {
        if (origImage.isValid() &&
            procImage.isValid()) {
            const float amount = (float)brightnessSlider.getValue();
```



```
        if (amount == 0.f) {
            procImage = origImage.createCopy();
        } else {
            for (int v = 0; v < origImage.getHeight(); ++v) {
                for (int h = 0; h < origImage.getWidth(); ++h) {
                    Colour col = origImage.getPixelAt (h, v);

                    if (amount > 0.f)
                        procImage.setPixelAt (h, v, col.brighter (amount));
                    else if (amount < 0.f)
                        procImage.setPixelAt (h, v, col.darker (-amount));
                }
            }
        }

        imageComponent.repaint();
    }
}
```

Here, we keep a copy of the original image and a processed version. Each time the slider changes, the image is updated with the new brightness by iterating over each of the pixels. When the **Write Image File...** button is clicked, we create a `FileChooser` object and present this to the user with the `FileChooser::browseForFileToSave()` function, rather than the `FileChooser::browseForFileToOpen()` function as we did for reading the file. Then the `PNGImageFormat` class is used to write the processed image to the selected file as a file stream. The image processing here could be significantly optimized, but that is beyond the scope of this book.

## Playing audio files

JUCE provides a sophisticated set of classes for dealing with audio. This includes: sound file reading and writing utilities, interfacing with the native audio hardware, audio data conversion functions, and a cross-platform framework for creating audio plugins for a range of well-known host applications. Covering all of these aspects is beyond the scope of this book, but the examples in this section will outline the principles of playing sound files and communicating with the audio hardware. In addition to showing the audio features of JUCE, in this section we will also create the GUI and autogenerate some other aspects of the code using the `IntroJucer` application.

## Creating a GUI to control audio file play

Create a new GUI application Introjucer project named `Chapter04_04`, selecting the option to create a basic window. In the Introjucer application, select the **Config** panel, and select **Modules** in the hierarchy.

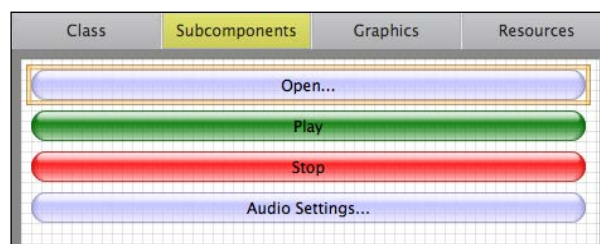
For this project we need the `juce_audio_utils` module (which contains a special Component class for configuring the audio device hardware); therefore, turn ON this module. Even though we created a basic window and a basic component, we are going to create the GUI using the Introjucer application in a similar way to that at the end of *Chapter 2, Building User Interfaces*.

Navigate to the **Files** panel and right-click (on the Mac, press *control* and click) on the **Source** folder in the hierarchy, and select **Add New GUI Component...** from the contextual menu.


When asked, name the header `MediaPlayer.h` and click on **Save**. In the **Files** hierarchy, select the `MediaPlayer.cpp` file. First select the **Class** panel and change the **Class name** from `NewComponent` to `MediaPlayer`. We will need four buttons for this basic project: a button to open an audio file, a **Play** button, a **Stop** button, and an audio device settings button. Select the **Subcomponents** panel, and add four `TextButton` components to the editor by right-clicking to access the contextual menu. Space the buttons equally near the top of the editor, and configure each button as outlined in the table as follows:

Purpose	member name	name	text	background (normal)
Open file	<code>openButton</code>	<code>open</code>	Open...	Default
Play/pause file	<code>playButton</code>	<code>play</code>	Play	Green
Stop playback	<code>stopButton</code>	<code>stop</code>	Stop	Red
Configure audio	<code>settingsButton</code>	<code>settings</code>	Audio Settings...	Default

Arrange the buttons as shown in the following screenshot:



For each button, access the **mode** pop-up menu for the **width** setting, and choose **Subtracted from width of parent**. This will keep the right-hand side of the buttons the same distance from the right-hand side of the window if the window is resized. There are more customizations to be done in the Introjucer project, but for now, make sure that you have saved the `MediaPlayer.h` file, the `MediaPlayer.cpp` file, and the Introjucer project before you open your native IDE project.

 Make sure that you have saved all of these files in the Introjucer application; otherwise the files may not get correctly updated in the file system when the project is opened in the IDE.

In the IDE we need to replace the `MainContentComponent` class code to place a `MediaPlayer` object within it. Change the `MainComponent.h` file as follows:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void resized();

private:
    MediaPlayer player;
};
#endif
```

Then, change the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    addAndMakeVisible (&player);
    setSize (player.getWidth(),player.getHeight());
}

void MainContentComponent::resized()
{
    player.setBounds (0, 0, getWidth(), getHeight());
}
```

Finally, make the window resizable in the `Main.cpp` file (as we did in the *Adding child components* section of *Chapter 2, Building User Interfaces*), and build and run the project to check that the window appears as expected.

## Adding audio file playback support

Quit the application and return to the `Introjucer` project. Select the `MediaPlayer.cpp` file in the **Files** panel hierarchy and select its **Class** panel. The **Parent classes** setting already contains `public Component`. We are going to be listening for state changes from two of our member objects that are `ChangeBroadcaster` objects. To do this, we need our `MediaPlayer` class to inherit from the `ChangeListener` class. Change the **Parent classes** setting such that it reads:

```
public Component, public ChangeListener
```

Save the `MediaPlayer.h` file, the `MediaPlayer.cpp` file, and the `Introjucer` project again, and open it into your IDE. Notice in the `MediaPlayer.h` file that the parent classes have been updated to reflect this change. For convenience, we are going to add some enumerated constants to reflect the current playback state of our `MediaPlayer` object, and a function to centralize the change of this state (which will, in turn, update the state of various objects, such as the text displayed on the buttons). The `ChangeListener` class also has one pure virtual function, which we need to add. Add the following code to the `[UserMethods]` section of `MediaPlayer.h`:

```
//[UserMethods] -- You can add your own custom methods...
enum TransportState {
    Stopped,
    Starting,
    Playing,
    Pausing,
    Paused,
    Stopping
};
void changeState (TransportState newState);
void changeListenerCallback (ChangeBroadcaster* source);
//[UserMethods]
```

We also need some additional member variables to support our audio playback. Add these to the `[UserVariables]` section:

```
//[UserVariables] -- You can add your own custom variables...
AudioDeviceManager deviceManager;
AudioFormatManager formatManager;
ScopedPointer<AudioFormatReaderSource> readerSource;
AudioTransportSource transportSource;
AudioSourcePlayer sourcePlayer;
TransportState state;
//[UserVariables]
```

The `AudioDeviceManager` object will manage our interface between the application and the audio hardware. The `AudioFormatManager` object will assist in creating an object that will read and decode the audio data from an audio file. This object will be stored in the `ScopedPointer<AudioFormatReaderSource>` object. The `AudioTransportSource` object will control the playback of the audio file and perform any sampling rate conversion that may be required (if the sampling rate of the audio file differs from the audio hardware sampling rate). The `AudioSourcePlayer` object will stream audio from the `AudioTransportSource` object to the `AudioDeviceManager` object. The state variable will store one of our enumerated constants to reflect the current playback state of our `MediaPlayer` object.

Now add some code to the `MediaPlayer.cpp` file. In the `[Constructor]` section of the constructor, add following two lines:

```
playButton->setEnabled (false);  
stopButton->setEnabled (false);
```

This sets the **Play** and **Stop** buttons to be disabled (and grayed out) initially. Later, we enable the **Play** button once a valid file is loaded, and change the state of each button and the text displayed on the buttons, depending on whether the file is currently playing or not. In this `[Constructor]` section you should also initialize the `AudioFormatManager` as follows:

```
formatManager.registerBasicFormats();
```

This allows the `AudioFormatManager` object to detect different audio file formats and create appropriate file reader objects. We also need to connect the `AudioSourcePlayer`, `AudioTransportSource` and `AudioDeviceManager` objects together, and initialize the `AudioDeviceManager` object. To do this, add the following lines to the `[Constructor]` section:

```
sourcePlayer.setSource (&transportSource);  
deviceManager.addAudioCallback (&sourcePlayer);  
deviceManager.initialise (0, 2, nullptr, true);
```

The first line connects the `AudioTransportSource` object to the `AudioSourcePlayer` object. The second line connects the `AudioSourcePlayer` object to the `AudioDeviceManager` object. The final line initializes the `AudioDeviceManager` object with:

- The number of required audio input channels (0 in this case).
- The number of required audio output channels (2 in this case, for stereo output).

- An optional "saved state" for the `AudioDeviceManager` object (`nullptr` initializes from scratch).
- Whether to open the default device if the saved state fails to open. As we are not using a saved state, this argument is irrelevant, but it is useful to set this to `true` in any case.

The final three lines to add to the [Constructor] section to configure our `MediaPlayer` object as a listener to the `AudioDeviceManager` and `AudioTransportSource` objects, and sets the current state to `Stopped`:

```
deviceManager.addChangeListener (this);
transportSource.addChangeListener (this);
state = Stopped;
```

In the `buttonClicked()` function we need to add some code to the various sections. In the [UserCode\_openButton] section, add:

```
//[UserCode_openButton] -- add your button handler...
FileChooser chooser ("Select a Wave file to play...",
                    File::nonexistent,
                    "*.wav");

if (chooser.browseForFileToOpen()) {
    File file (chooser.getResult());
    readerSource = new AudioFormatReaderSource(
        formatManager.createReaderFor (file), true);
    transportSource.setSource (readerSource);
    playButton->setEnabled (true);
}
//[UserCode_openButton]
```

When the `openButton` button is clicked, this will create a `FileChooser` object that allows the user to select a file using the native interface for the platform. The types of files that are allowed to be selected are limited using the wildcard `*.wav` to allow only files with the `.wav` file extension to be selected.

If the user actually selects a file (rather than cancels the operation), the code can call the `FileChooser::getResult()` function to retrieve a reference to the file that was selected. This file is then passed to the `AudioFormatManager` object to create a file reader object, which in turn is passed to create an `AudioFormatReaderSource` object that will manage and own this file reader object. Finally, the `AudioFormatReaderSource` object is connected to the `AudioTransportSource` object and the **Play** button is enabled.

The handlers for the `playButton` and `stopButton` objects will make a call to our `changeState()` function depending on the current transport state. We will define the `changeState()` function in a moment where its purpose should become clear.

In the `[UserButtonCode_playButton]` section, add the following code:

```
//[UserButtonCode_playButton] -- add your button handler...
if ((Stopped == state) || (Paused == state))
    changeState (Starting);
else if (Playing == state)
    changeState (Pausing);
//[UserButtonCode_playButton]
```

This changes the state to `Starting` if the current state is either `Stopped` or `Paused`, and changes the state to `Pausing` if the current state is `Playing`. This is in order to have a button with combined play and pause functionality.

In the `[UserButtonCode_stopButton]` section, add the following code:

```
//[UserButtonCode_stopButton] -- add your button handler...
if (Paused == state)
    changeState (Stopped);
else
    changeState (Stopping);
//[UserButtonCode_stopButton]
```

This sets the state to `Stopped` if the current state is `Paused`, and sets it to `Stopping` in other cases. Again, we will add the `changeState()` function in a moment, where these state changes update various objects.

In the `[UserButtonCode_settingsButton]` section add the following code:

```
//[UserButtonCode_settingsButton] -- add your button handler...
bool showMidiInputOptions = false;
bool showMidiOutputSelector = false;
bool showChannelsAsStereoPairs = true;
bool hideAdvancedOptions = false;

AudioDeviceSelectorComponent settings (deviceManager,
                                       0, 0, 1, 2,
                                       showMidiInputOptions,
                                       showMidiOutputSelector,
                                       showChannelsAsStereoPairs,
                                       hideAdvancedOptions);

settings.setSize (500, 400);
```

---

```

DialogWindow::showModalDialog(String ("Audio Settings"),
                                &settings,
                                TopLevelWindow::getTopLevelWindow (0),
                                Colours::white,
                                true); //[/UserCode_settingsButton]

```

This presents a useful interface to configure the audio device settings.

We need to add the `changeListenerCallback()` function to respond to changes in the `AudioDeviceManager` and `AudioTransportSource` objects. Add the following to the `[MiscUserCode]` section of the `MediaPlayer.cpp` file:

```

//[/MiscUserCode] You can add your own definitions...
void MediaPlayer::changeListenerCallback (ChangeBroadcaster* src)
{
    if (&deviceManager == src) {
        AudioDeviceManager::AudioDeviceSetup setup;
        deviceManager.getAudioDeviceSetup (setup);

        if (setup.outputChannels.isZero())
            sourcePlayer.setSource (nullptr);
        else
            sourcePlayer.setSource (&transportSource);
    } else if (&transportSource == src) {
        if (transportSource.isPlaying()) {
            changeState (Playing);
        } else {
            if ((Stopping == state) || (Playing == state))
                changeState (Stopped);
            else if (Pausing == state)
                changeState (Paused);
        }
    }
}
//[/MiscUserCode]

```

If our `MediaPlayer` object receives a message that the `AudioDeviceManager` object changed in some way, we need to check that this change wasn't to disable all of the audio output channels, by obtaining the setup information from the device manager. If the number of output channels is zero, we disconnect our `AudioSourcePlayer` object from the `AudioTransportSource` object (otherwise our application may crash) by setting the source to `nullptr`. If the number of output channels becomes nonzero again, we reconnect these objects.



If our `AudioTransportSource` object has changed, this is likely to be a change in its playback state. It is important to note the difference between requesting the transport to start or stop, and this change actually taking place. This is why we created the enumerated constants for all the other states (including transitional states). Again we issue calls to the `changeState()` function depending on the current value of our state variable and the state of the `AudioTransportSource` object.

Finally, add the important `changeState()` function to the `[MiscUserCode]` section of the `MediaPlayer.cpp` file that handles all of these state changes:

```
void MediaPlayer::changeState (TransportState newState)
{
    if (state != newState) {
        state = newState;
        switch (state) {
            case Stopped:
                playButton->setButtonText ("Play");
                stopButton->setButtonText ("Stop");
                stopButton->setEnabled (false);
                transportSource.setPosition (0.0);
                break;
            case Starting:
                transportSource.start();
                break;
            case Playing:
                playButton->setButtonText ("Pause");
                stopButton->setButtonText ("Stop");
                stopButton->setEnabled (true);
                break;
            case Pausing:
                transportSource.stop();
                break;
            case Paused:
                playButton->setButtonText ("Resume");
                stopButton->setButtonText ("Return to Zero");
                break;
            case Stopping:
                transportSource.stop();
                break;
        }
    }
}
```

After checking that the `newState` value is different from the current value of the state variable, we update the state variable with the new value. Then, we perform the appropriate actions for this particular point in the cycle of state changes. These are summarized as follows:

- In the `Stopped` state, the buttons are configured with the **Play** and **Stop** labels, the **Stop** button is disabled, and the transport is positioned to the start of the audio file.
- In the `Starting` state, the `AudioTransportSource` object is told to start. Once the `AudioTransportSource` object has actually started playing, the system will be in the `Playing` state. Here we update the `playButton` button to display the text **Pause**, ensure the `stopButton` button displays the text **Stop**, and we enable the **Stop** button.
- If the **Pause** button is clicked, the state becomes `Pausing`, and the transport is told to stop. Once the transport has actually stopped, the state changes to `Paused`, the `playButton` button is updated to display the text **Resume** and the `stopButton` button is updated to display **Return to Zero**.
- If the **Stop** button is clicked, the state is changed to `Stopping`, and the transport is told to stop. Once the transport has actually stopped, the state changes to `Stopped` (as described in the first point).
- If the **Return to Zero** button is clicked, the state is changed directly to `Stopped` (again, as previously described).
- When the audio file reaches the end of the file, the state is also changed to `Stopped`.

Build and run the application. You should be able to select a `.wav` audio file after clicking the **Open...** button, play, pause, resume, and stop the audio file using the respective buttons, and configure the audio device using the **Audio Settings...** button. The audio settings window allows you to select the input and output device, the sample rate, and the hardware buffer size. It also provides a **Test** button that plays a tone through the selected output device.

## Working with the Binary Builder tool

One problem with writing cross-platform applications is the packaging of binary files for use within the application. JUCE includes the **Binary Builder** tool that transforms binary files into source code, which is then compiled into the application's code. This ensures that the files will behave identically on all platforms, rather than relying on peculiarities of the runtime machine. Although the Binary Builder is available as a separate project (in `juce/extras/binarybuilder`), its functionality is available within the Introjucer application's GUI component editor.

## Embedding an image file using the Introjucer application

Create a new Introjucer project named `Chapter04_05` with a basic window. Add a new GUI component as before; this time name it `EmbeddedImage` (remembering to also change the name in its **Class** panel). In its **Subcomponents** panel, right-click in the canvas and choose **New Generic Component** and resize it to fill the canvas with a small border around the edge. Change the **member name** and **name** to `image`, and change the **class** to `ImageComponent`. In the **Resources** panel, choose **Add new resource...** and select an image file to add. This will create a resource that is the binary file converted to code. It will be given a variable name within this component based on the original filename, and will be stored as a static variable. For example, a file named `sample.png` will be named `sample_png`. In addition to this a static variable storing this resource's size as an integer will be created and will have `Size` appended to this name, for example, `sample_pngSize`. Save the project and open it into your IDE. Update the `MainComponent` file's contents as before. Change the `MainComponent.h` file as follows:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void resized();

private:
    EmbeddedImage embeddedImage;
};
#endif
```

Then change the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    addAndMakeVisible (&embeddedImage);
    setSize (embeddedImage.getWidth(), embeddedImage.getHeight());
}

void MainContentComponent::resized()
{
    embeddedImage.setBounds (0, 0, getWidth(), getHeight());
}
```

Finally in the `EmbeddedImage.cpp` file notice the large arrays of numbers at the end of the file, this is the image file converted to code. In the `[Constructor]` section, add the following two lines (although you may need to use different names from `sample_png`, `sample_pngSize`, depending on the file resource you added previously):

```
//[Constructor] You can add your own custom stuff here...
MemoryInputStream stream (sample_png, sample_pngSize, false);
image->setImage (ImageFileFormat::loadFrom (stream));
//[Constructor]
```

This creates a memory stream from our resource, providing the data pointer and the data size (the final `false` argument tells the memory stream not to copy the data). Then we load the image as before using the `ImageFileFormat` class. Build and run the application, and the image should be displayed into the application's window.

## Summary

This chapter has covered a range of techniques for dealing with files in JUCE, focusing in particular on image and audio files. You are encouraged to explore the online JUCE documentation, which provides even more detail on many of the possibilities introduced here. We have also introduced the Binary Builder tool that provides a means of transforming media files into source code that is suitable for cross-platform use. You are encouraged to read the online JUCE documentation for each of the classes introduced in this chapter. This chapter has given only an introduction to get you started; there are many other options and alternative approaches, which may suit different circumstances. The JUCE documentation will take you through each of these and point you to related classes and functions. The next chapter covers some useful utilities available within JUCE for creating cross-platform applications.

# 5

## Helpful Utilities

In addition to the essential classes introduced in the previous chapters, JUCE includes a range of classes for solving common problems in application development. In this chapter we will cover the following topics:

- Using the `Value`, `var`, and `ValueTree` classes
- Implementing undo management
- Adding XML support
- Understanding how JUCE handles multiple threads
- Storing the application properties
- Adding the menu bar controls

By the end of this chapter, you will have an awareness of some of the additional helpful utilities offered by JUCE.

## Using the dynamically typed objects

The JUCE `Value`, `var`, and `ValueTree` classes are valuable tools for application data storage and handling. The `var` class (short for variant) is designed to store a range of primitive data types including integers, floating-point numbers, and strings (JUCE `String` objects). It may also be recursive in the sense that a `var` instance can hold an array of `var` instances (a JUCE `Array<var>` object). In this way, the `var` class is similar to the dynamic types supported by many scripting languages such as JavaScript. A `var` object may also hold a reference to any kind of `ReferenceCounterObject` object or chunks of binary data. All of the following are valid initializations:

```
var anInt = 1;
var aDouble = 1.2345;
var aString = "Hello world!";
```

## Using the Value class

The `Value` class is designed to hold a shared instance of a `var` object (by storing the `var` in a reference-counted wrapper). A `Value` object may have listeners attached using a `Value::Listener` function and the same techniques covered in the *Chapter 2, Building User Interfaces* regarding the listener and broadcaster system employed by the GUI classes. In fact, `Value` objects are used by the various `Component` subclasses to store any value, such as the text in a `Label` object, the position of the thumb in a `Slider` object, and so on. As an example, the following code snippets illustrate ways of setting a `Label` object's value and a `Slider` object's value using its `Value` object:

```
// Slider
Slider slider;
slider.getValueObject().setValue (10.0);
// instead of:
// slider.setValue (10);

// Label
Label label;
label.getTextValue().setValue ("Hello");
// instead of:
// label.setText ("Hello", sendNotification);
```

The value objects are also a way of sharing values, because they can be made to refer to the same underlying data. This can be useful where the same value is displayed in a GUI in different ways, especially in complex and detailed GUI displays. Create a new Introjucer project named `Chapter05_01` with a basic window, and replace the `MainComponent.h` file with the following:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void resized();

private:
    Value value;
    Slider slider;
    Label label;
};

#endif // __MAINCOMPONENT_H__
```

Here we store the `Value`, `Slider`, and `Label` objects in our class. Replace the `MainComponent.cpp` file with the following:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: value (1.0),
  slider (Slider::LinearHorizontal, Slider::NoTextBox)
{
    label.setEditable (true);
    slider.getValueObject().referTo (value);
    label.getTextValue().referTo (value);

    addAndMakeVisible (&slider);
    addAndMakeVisible (&label);

    setSize (500, 400);
}

void MainContentComponent::resized()
{
    slider.setBounds (10, 10, getWidth() - 20, 20);
    label.setBounds (10, 40, getWidth() - 20, 20);
}
```



Here, we initialize our `Value` object to the value of one, then use the `Value::referTo()` function to configure both the `Slider` and the `Label` objects' values to refer to this same underlying `Value` object. Build and run the application noticing that both the slider and label keep updated with the same value regardless of which one is changed. This is all achieved without us needing to configure our own listeners since JUCE handles all of this internally.

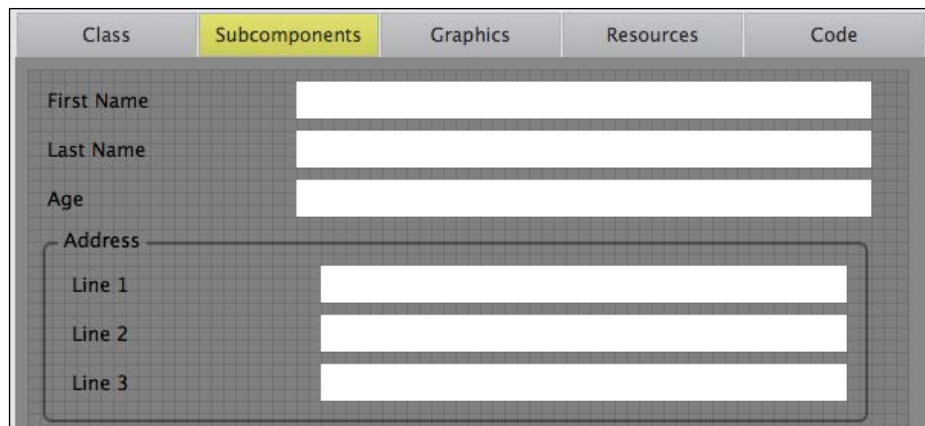
## Structuring hierarchical data

Clearly, in most applications the data model is much more complex and commonly hierarchical. The `ValueTree` class is designed to reflect this using a relatively lightweight, yet powerful implementation. A `ValueTree` object holds a tree structure of named `var` objects as properties, meaning that nodes in the tree can be almost any data type. The following example illustrates how to store data in a `ValueTree` object, and some of the features that make the `ValueTree` class so invaluable to JUCE application development.

Create a new IntroJucer project named `Chapter05_02` with a basic window. Add a GUI component named `EntryForm` in a similar way to previous chapters. First, navigate to the **Graphics** panel for the `EntryForm.cpp` file, and change the background color to gray. Now we will add a form-like page into which we can enter a person's name, age, and address. Add six `Label` objects to the **Subcomponents** panel to act as labels for the data with the following contents: **First name**, **Last name**, **Age**, **Line 1**, **Line 2**, and **Line 3**.

Now add six `Label` objects with no contents (that is, empty text) adjacent to each of the labels added in the previous step. Set these to have a white, rather than transparent background, and set their **editing** property to **edit on single-click**. Give these the following **member name**, and **name** values as follows: `firstNameField`, `lastNameField`, `ageField`, `line1Field`, `line2Field`, and `line3Field`.

Finally, add a **Group Box** via the contextual menu accessed by right-clicking (on the Mac, press *control* and click) in the **Subcomponents** editor. Position this to surround the labels related to **Line 1**, **Line 2**, **Line 3**, and their entry fields. This should now look similar to the following screenshot:



Now save the project and open it in your IDE. Place an `EntryForm` object into the `MainContentComponent` object as before, by changing the `MainComponent.h` file to contain the following:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component
{
public:
    MainContentComponent();
    void resized();

private:
    EntryForm form;
};
#endif
```

Change the `MainComponent.cpp` file to contain the following:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    addAndMakeVisible (&form);
    setSize (form.getWidth(), form.getHeight());
}

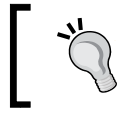
void MainContentComponent::resized()
{
    form.setBounds (0, 0, getWidth(), getHeight());
}
```

We now need to add some custom code to the `EntryForm` class to get it to store its data in a `ValueTree` object. First, add some variables to the class in the `EntryForm.h` file in the `[UserVariables]` section as follows:

```
//[UserVariables]    -- You can add your own custom variables..
ValueTree personData;

static const Identifier personId;
static const Identifier firstNameId;
static const Identifier lastNameId;
static const Identifier ageId;
static const Identifier addressId;
static const Identifier line1Id;
static const Identifier line2Id;
static const Identifier line3Id;
//[UserVariables]
```

Here we have one `ValueTree` object that will store the data and several static `Identifier` objects (which we will initialize in the `EntryForm` class in a moment) that serve as names for the `ValueTree` object structure and its properties. An `Identifier` object is effectively a special type of `String` object that holds only a limited set of characters, such that it will be valid in other contexts (for example, variable names, XML).



It is more efficient to create these Identifier objects when the application starts rather than create them each time they are needed.

In the `EntryForm.cpp` file add the following code to initialize the Identifier objects to the `[MiscUserCode]` section:

```
//[MiscUserCode] You can add your own definitions ...
const Identifier EntryForm::personId = "person";
const Identifier EntryForm::firstNameId = "firstName";
const Identifier EntryForm::lastNameId = "lastName";
const Identifier EntryForm::ageId = "age";
const Identifier EntryForm::addressId = "address";
const Identifier EntryForm::line1Id = "line1";
const Identifier EntryForm::line2Id = "line2";
const Identifier EntryForm::line3Id = "line3";
//[MiscUserCode]
```

In the constructor we need to initialize the ValueTree object, so add the following code to the `[Constructor]` section:

```
//[Constructor] You can add your own custom stuff here..
personData = ValueTree (personId);
personData.setProperty (firstNameId, String::empty, nullptr);
personData.setProperty (lastNameId, String::empty, nullptr);
personData.setProperty (ageId, String::empty, nullptr);

ValueTree addressData = ValueTree (addressId);
addressData.setProperty (line1Id, String::empty, nullptr);
addressData.setProperty (line2Id, String::empty, nullptr);
addressData.setProperty (line3Id, String::empty, nullptr);
personData.addChild (addressData, -1, nullptr);
//[Constructor]
```

Here we create the main `ValueTree` object with the named type `person`, and add three properties for the first name, last name, and age. (The graphic layout of our `EntryForm` component indicates the hierarchical relationship between the values in this top-level `personData` object.) The `nullptr` argument in each case indicates that we do not want undo management; we will look at this later in the chapter. We then create another `ValueTree` object with the named type `address`. Then, we add the three lines of the address as properties and add it as a **child node** to the main `ValueTree` object. It is in this way that we create tree structures with several `ValueTree` objects. The second argument of the call to the `ValueTree::addChild()` function indicates the index at which we want to add the child node. The `-1` parameter passed in this case indicates we just want to add it to the end of the list of nodes (but we have only one in any case; therefore, this value is of little importance).

Finally, we need to update the `ValueTree` object when the labels change. Add the following code the appropriate sections:

```
//[UserLabelCode_firstNameField] -- add your label text handling..
personData.setProperty (firstNameId,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_firstNameField]
...
//[UserLabelCode_lastNameField] -- add your label text handling..
personData.setProperty (lastNameId,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_lastNameField]
...
//[UserLabelCode_ageField] -- add your label text handling..
personData.setProperty (ageId,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_ageField]
...
//[UserLabelCode_line1Field] -- add your label text handling..
ValueTree addressData (personData.getChildWithName (addressId));
addressData.setProperty (line1Id,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_line1Field]
...
//[UserLabelCode_line2Field] -- add your label text handling..
ValueTree addressData (personData.getChildWithName (addressId));
addressData.setProperty (line2Id,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_line2Field]
...
//[UserLabelCode_line3Field] -- add your label text handling..
ValueTree addressData (personData.getChildWithName (addressId));
addressData.setProperty (line3Id,
                        labelThatHasChanged->getText(), nullptr);
//[UserLabelCode_line3Field]
```

Build and run the application and confirm that you can edit the contents of the entry form fields. You may also notice that JUCE automatically implements **focus ordering**, such that you can use the *Tab* key to move between fields. However, this isn't very useful yet since we don't do anything with the data. In the next section we will add undo management, which will start to show the power of the `ValueTree` class.

## Employing undo management

JUCE includes an `UndoManager` class to help manage undo and redo actions. This can be used independently, but works almost automatically if the application's data is stored in a `ValueTree` object. To illustrate this we need to make a few changes to the project developed so far. First make some changes in the Introjucer project. Add a `TextButton` subcomponent labeled **Undo** and change its **name** and **member name** to `undoButton`. In the **Class** panel add the `ValueTree::Listener` class to the **Parent classes** property such that it reads:

```
public Component, public ValueTree::Listener
```

Save all the files and the project, and open it into your IDE. Add the following code for the `ValueTree::Listener` class to the `[UserMethods]` section of the `MainComponent.h` file. Notice that we add empty function braces except for the `valueTreePropertyChanged()` function here since we do not need to add code for the other functions:

```
//[UserMethods]      -- You can add your own custom methods ...
void valueTreePropertyChanged (ValueTree& tree,
                               const Identifier& property);
void valueTreeChildAdded (ValueTree& parentTree,
                          ValueTree& child)      { }
void valueTreeChildRemoved (ValueTree& parentTree,
                           ValueTree& child)    { }
void valueTreeChildOrderChanged (ValueTree& tree) { }
void valueTreeParentChanged (ValueTree& tree)   { }
void valueTreeRedirected (ValueTree& tree)      { }
//[UserMethods]
```

Add an `UndoManager` object to the `[UserVariables]` section as follows:

```
//[UserVariables]    -- You can add your own custom variables...
UndoManager undoManager;
...
```

In the `EntryForm.cpp` file add the following code for the `ValueTree::Listener` functions to the `[MiscUserCode]` section (notice that we need only one of these functions as we added empty functions to the preceding header file):

```
void EntryForm::valueTreePropertyChanged
(ValueTree& tree, const Identifier& property)
{
    if (property == firstNameId) {
        firstNameField->setText (tree.getProperty (property),
                                dontSendNotification);
    } else if (property == lastNameId) {
        lastNameField->setText (tree.getProperty (property),
                                dontSendNotification);
    } else if (property == ageId) {
        ageField->setText (tree.getProperty (property),
                           dontSendNotification);
    } else if (property == line1Id) {
        line1Field->setText (tree.getProperty (property),
                             dontSendNotification);
    } else if (property == line2Id) {
        line2Field->setText (tree.getProperty (property),
                             dontSendNotification);
    } else if (property == line3Id) {
        line3Field->setText (tree.getProperty (property),
                             dontSendNotification);
    }
}
```

Add our `EntryForm` object as a listener to the main value tree by adding the following code to the end of the `[Constructor]` section:

```
...
personData.addListener (this);
//[Constructor]
```

Each time we call the `ValueTree::setProperty()` function we need to pass a pointer to our `UndoManager` object. Find each line of code that uses `ValueTree::setProperty()`, and change the `nullptr` argument to `&undoManager`, for example:

```
//[UserLabelCode_firstNameField] -- add your label text handling..
personData.setProperty (firstNameId,
                        labelThatHasChanged->getText(),
                        &undoManager);
//[UserLabelCode_firstNameField]
```

Do not use a simple find-and-replace since there are other uses of `nullptr` in the code that do not relate to the `ValueTree` object and `UndoManager` object code. In our application, when we make a change that we want to be undoable, we need to tell the `UndoManager` object what comprises a **transaction**. In some cases it might be appropriate to consider each minor change as a transaction. In other cases it might be more useful to the user to group small changes into a single transaction (for example, changes that occur within a certain time limit, or multiple changes to the same object). We will make each of the changes in the `EntryForm::labelTextChanged()` function a transaction, so add the following code to the `[UserlabelTextChanged_Pre]` section:

```
//[UserlabelTextChanged_Pre]
undoManager.beginNewTransaction();
//[UserlabelTextChanged_Pre]
```

Finally, execute the undo action in the `[UserButtonCode_undoButton]` section by adding the following code:

```
//[UserButtonCode_undoButton] -- add your button handler..
undoManager.undo();
//[UserButtonCode_undoButton]
```

This line tells the `UndoManager` object to undo the last transaction. Adding redo support is just as straightforward. Build and run the application, and notice that you can now undo changes to the data entry form using the **Undo** button. The `ValueTree` class also supports serialization and deserialization via binary or XML formats; this will be outlined in the next section.



## Adding XML support

JUCE includes a range of support for XML parsing and storage. You may have noticed that the Introjucer application uses the XML format to store metadata at the end of some of the autogenerated files (for example, in our `EntryForm.cpp` file). In particular, a `ValueTree` object can be serialized into XML, and this same XML can be deserialized back into a `ValueTree` object (although you can't convert any arbitrary XML to a `ValueTree` object without doing some of your own parsing). To add opening and saving capabilities to our project, first we need to add an **Open...** and a **Save...** button in the Introjucer project. Give these the **name** and **member name** `openButton` and `saveButton` respectively. Then, in the code we need to perform the conversions to and from XML. In the `[UserButtonCode_saveButton]` section add the following code to present the user with a file chooser and save the `ValueTree` object's data to an XML file:

```
//[UserButtonCode_saveButton] -- add your button handler...
FileChooser chooser ("Save person data",
                    File::nonexistent,
                    "*.xml");

if (chooser.browseForFileToSave (true)) {
    File file (chooser.getResult());

    if (file.existsAsFile())
        file.moveToTrash();

    FileOutputStream stream (file);

    ScopedPointer<XmlElement> xml = personData.createXml();
    xml->writeToStream (stream, String::empty);
}
//[UserButtonCode_saveButton]
```

In the [UserButtonCode\_openButton] section add the following code to read an XML file back in to the ValueTree object:

```
//[UserButtonCode_openButton] -- add your button handler...
FileChooser chooser ("Open person data",
                    File::nonexistent,
                    "*.xml");

if (chooser.browseForFileToOpen()) {
    Logger* log = Logger::getCurrentLogger();
    File file (chooser.getResult());

    XmlDocument xmlDoc (file);
    ScopedPointer<XmlElement> xml = xmlDoc.getDocumentElement();

    if (xml == nullptr) {
        log->writeToLog ("XML error");
        return;
    }

    ValueTree newPerson (ValueTree::fromXml (*xml));

    if (newPerson.getType() != personId) {
        log->writeToLog ("Invalid person XML");
        return;
    }

    undoManager.beginNewTransaction();
    personData.copyPropertiesFrom (newPerson, &undoManager);

    ValueTree newAddress (newPerson.getChildWithName (addressId));
    ValueTree addressData (personData.getChildWithName (addressId));
    addressData.copyPropertiesFrom (newAddress, &undoManager);
}
//[UserButtonCode_openButton]
```

Here we load the chosen file as an XML document, and access its document element. We perform two checks on the XML and report an error to the log if necessary:

- We check if the XML element was accessed successfully (that is, did not return `nullptr`). If this fails the file may not be a valid XML file.
- We load the XML into a `ValueTree` object then check the type of this `ValueTree` object to ensure it is the person data that we expect.

Once the loaded `ValueTree` object is checked successfully, we copy the properties to the stored `ValueTree` object as a single `UndoManager` object transaction.

Build and run the application, and check that saving, opening, and all the undo behaviors work as expected. The following screenshot shows how the application window should appear:



The XML file produced by this code for the data shown in this screenshot will look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<person firstName="Joe" lastName="Bloggs" age="25">
  <address line1="1 The Lines" line2="Loop" line3="Codeland"/>
</person>
```

Of course, in a real-world application we would have **Open**, **Save**, and **Undo** commands as menu bar items too (or instead), but we have used buttons here for simplicity. (Adding menu bar controls is covered at the end of this chapter.)

The `XmlDocument` and `XmlElement` classes shown here provide broad functionality for parsing and creating XML documents independent of `ValueTree` objects.

## Understanding how JUCE handles multiple threads

JUCE includes a cross-platform interface to operating system threads using its `Thread` class. There are also classes that help with synchronizing inter-thread communication, notably the `CriticalSection` class, the `WaitableEvent` class, and the `Atomic` template classes (for example, `Atomic<int>`). Writing multithreaded applications is inherently challenging and it is beyond the scope of this book to serve as an introduction. However, JUCE does make the processes of writing multithreaded applications a little easier. One way in which this is achieved is through providing a consistent interface on all platforms. JUCE will also raise assertions if you do certain things that are likely to lead to some of the common problems (for example, deadlocks and race conditions). The following serves as a basic demonstration; we will create a simple thread that increments a counter and displays this counter in the GUI. Create a new Introjucer project called `Chapter05_03` with a basic window. Open the project in your IDE, and change the `MainComponent.h` file to contain the following:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

class MainContentComponent : public Component,
                             public Button::Listener,
                             public Thread
{
public:
    MainContentComponent();
    ~MainContentComponent();

    void resized();
    void buttonClicked (Button* button);
    void run();

private:
    TextButton startThreadButton;
    TextButton stopThreadButton;
    Label counterLabel;
    int counter;
};
#endif // __MAINCOMPONENT_H__
```

Notice that our class inherits from the `Thread` class that requires us to implement the `Thread::run()` pure virtual function (which serves as our thread's entry point). Now replace the code in the `MainComponent.cpp` file with the following:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
: Thread ("Counter Thread"),
  startThreadButton ("Start Thread"),
  stopThreadButton ("Stop Thread"),
  counter (0)
{
    addAndMakeVisible (&startThreadButton);
    addAndMakeVisible (&stopThreadButton);
    addAndMakeVisible (&counterLabel);

    startThreadButton.addListener (this);
    stopThreadButton.addListener (this);

    setSize (500, 400);
}

MainContentComponent::~MainContentComponent()
{
    stopThread (3000);
}

void MainContentComponent::resized()
{
    startThreadButton.setBounds (10, 10, getWidth() - 20, 20);
    stopThreadButton.setBounds (10, 40, getWidth() - 20, 20);
    counterLabel.setBounds (10, 70, getWidth() - 20, 20);
}

void MainContentComponent::buttonClicked (Button* button)
{
    if (&startThreadButton == button)
        startThread();
    else if (&stopThreadButton == button)
        stopThread (3000);
}
```

The main thing to notice here is that we must provide a name for our thread by passing a `String` object to the `Thread` class constructor. In the `buttonClicked()` function we start and stop our thread using the **Start Thread** and **Stop Thread** buttons, respectively. The value of 3000 passed to the `Thread::stopThread()` function is a timeout in milliseconds, after which the thread will be forcibly killed (which is unlikely to happen unless there is an error). We also need to implement the `Thread::run()` function, which is where the thread undertakes its work. This is where many of the problems occur. In particular you can't directly update GUI objects from anything other than the JUCE message thread. This message thread is the main thread on which your application's `initialise()` and `shutdown()` functions are called (so most of the construction and destruction of your application), where your GUI listener callbacks are called, mouse events are reported, and so on. Effectively, it is the "main" thread (and probably is the executable's main thread in many circumstances). This is why it has been safe to update GUI objects in response to user interactions with other GUI objects. Add the following code to the end of the `MainComponent.cpp` file. This should fail as soon as you click on the **Start Thread** button:

```
void MainContentComponent::run()
{
    while (!threadShouldExit()) {
        counterLabel.setText (String (counter++),
                               dontSendNotification);
    }
}
```

In a *debug* build, your code should stop on an assertion. Looking at the JUCE code where the assertion was raised, there will be a comment that will tell you that this action can't be done unless you use a `MessageManagerLock` object. (In a *release* build, it may simply crash or cause the application to behave strangely.) To use a `MessageManagerLock` object correctly, change the `run()` function as follows:

```
void MainContentComponent::run()
{
    while (!threadShouldExit()) {
        const MessageManagerLock lock (Thread::getCurrentThread());

        if (lock.lockWasGained()) {
            counterLabel.setText (String (counter++),
                                   dontSendNotification);
        }
    }
}
```

Here we create a `MessageManagerLock` object, passing it a pointer to the current thread (that is, `this thread`). If the `MessageManagerLock::lockWasGained()` function returns `true`, it is safe to manipulate GUI objects. The thread releases the lock as the `MessageManagerLock` object goes out of scope (as we come round the `while()` loop again). This code also shows the typical structure of a `Thread::run()` function; that is, a `while()` loop that checks the result of calling the `Thread::threadShouldExit()` function and continues to loop unless the thread has been told to exit.

## Storing application properties

In this final example we will implement a simple application that stores its state in a properties file (that is, settings or preferences) in a standard location on the runtime platform. First create a new Introjucer project named `Chapter05_04` with a basic window. Change the `MainComponent.h` file to contain the following code:

```
#ifndef __MAINCOMPONENT_H__
#define __MAINCOMPONENT_H__

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

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

    void resized();

private:
    Label label;
    Slider slider;
    ApplicationProperties appProperties;
};

#endif // __MAINCOMPONENT_H__
```

Here we have a label and a slider; these will represent our simple application properties. Clearly, in a fully developed application, the properties would be presented in a separate window or panel, but the principle is the same.



The `ApplicationProperties` class is a helper class that manages the application properties, saving them to the appropriate location on the user's system.

Change the contents of the `MainComponent.cpp` file to:

```
#include "MainComponent.h"

MainContentComponent::MainContentComponent()
{
    label.setEditable(true);
    addAndMakeVisible(&label);
    addAndMakeVisible(&slider);

    setSize (500, 400);

    PropertiesFile::Options options;
    options.applicationName = ProjectInfo::projectName;
    options.folderName = ProjectInfo::projectName;
    options.filenameSuffix = "settings";
    options.osxLibrarySubFolder = "Application Support";
    appProperties.setStorageParameters (options);

    PropertiesFile* props = appProperties.getUserSettings();

    label.setText (props->getValue ("label", "<empty>"),
                  dontSendNotification);
    slider.setValue (props->getDoubleValue ("slider", 0.0));
}

MainContentComponent::~MainContentComponent()
{
    PropertiesFile* props = appProperties.getUserSettings();
    props->setValue ("label", label.getText());
    props->setValue ("slider", slider.getValue());
}

void MainContentComponent::resized()
{
    label.setBounds (10, 10, getWidth() - 20, 20);
    slider.setBounds (10, 40, getWidth() - 20, 20);
}
```



Here we configure the `ApplicationProperties` object by passing it our desired application name and folder name (using the name that the Introjucer application will have generated in the `ProjectInfo::projectName` constant). We provide a filename suffix (for example, `settings.xml`). To support Mac OS X, it is recommended that you set the `PropertiesFile::Options::osxLibrarySubFolder` option since Apple changed their recommendation for the storage of application preferences. This was previously in `Library/Preferences`, but Apple now recommends that developers use `Library/Application Support`. This is provided for backwards compatibility; all new applications should set this to `Application Support`. This setting is harmless for other platforms. It is important to configure these options prior to using the `ApplicationProperties` object by passing the options via the `ApplicationProperties::setStorageParameters()` function. In fact, the `ApplicationProperties` class maintains two sets of properties, one for all users and one for the current user. In this example we create only properties for the current user.

When the application starts, it tries to access the value of the properties and set the label and slider appropriately. First, we access the user settings `PropertiesFile` object using the `ApplicationProperties::getUserSettings()` function. We store a pointer to the `PropertiesFile` object that this returns in a regular pointer, as it is owned by `ApplicationProperties` object, and we need it only temporarily. (Storing it in a `ScopedPointer` object in this case could cause a crash, since the `ScopedPointer` object would try eventually to delete an object that it should not really own, since it already has an owner.) Then we use the `PropertiesFile::getValue()` function to get the text value, and the `PropertiesFile::getDoubleValue()` function to get the double value (there are also the `PropertiesFile::getIntValue()` and `PropertiesFile::getBoolValue()` functions if needed). Of course, the first time the application starts, these properties will be empty. Each of these property accessors allows you to provide a default value, should the named property not exist. Here we provide `<empty>` as the default for the label contents and `0.0` as the default for the slider. When the application closes (in this case we know this is happening when the `MainContentComponent` destructor is called) we set the value of the properties to the current state of the label and the slider. This means that when we close the application and reopen it, the slider and the label should appear to retain their state between launches. Build and run the application and test this. The file generated by the `ApplicationProperties` object should look something as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<PROPERTIES>
  <VALUE name="label" val="hello"/>
  <VALUE name="slider" val="1.62303665"/>
</PROPERTIES>
```

On Mac OS X this should be in:

```
~/Library/Application Support/Chapter05_04
```

On Windows this should be in:

```
C:\\Documents and Settings\\USERNAME\\Application Data\\Chapter05_04
```

where USERNAME is the name of the currently logged-in user.

## Adding menu bar controls

JUCE offers a means of creating menu bar user interface controls, as you will have seen, using the Introjucer application, and in the JUCE Demo application in *Chapter 1, Installing JUCE and the Introjucer Application*. These menu bars may be within a window on all platforms using JUCE's own `MenuBarComponent` class, or as a native menu bar at the top of the screen on Mac OS X. To demonstrate this we will add some special commands to the `Chapter05_04` project to reset the label and slider in various ways.

The first requirement for constructing menu bars in JUCE is to create a **menu bar model** by creating a subclass of the `MenuBarModel` class. First add the `MenuBarModel` class as a base class for the `MainContentComponent` class in the `MainComponent.h` file as highlighted in the following:

```
...
class MainContentComponent    : public Component,
                               public MenuBarModel
{
    ...
}
```

The `MenuBarModel` class has three pure virtual functions that will be used to populate the menu bar. To add these, add the following three lines to the `public` section of the `MainComponent.h` file:

```
StringArray getMenuBarNames();
PopupMenu getMenuForIndex (int index, const String& name);
void menuItemSelected (int menuID, int index);
```

The `getMenuBarNames()` function should return an array of menu names that will appear along the menu bar. The `getMenuForIndex()` function is used to create the actual menu when the user clicks on one of the menu bar names. This should return a `PopupMenu` object for a given menu (which can be determined using the menu index or its name). Each menu item should be given a unique ID value that is used to identify the menu item when it is selected. This is described in a moment. The `menuItemSelected()` function will be called when a user selects a particular menu item from one of the menus. Here you are provided with the ID value of the menu item that was selected (and the index of the menu that this menu item was in, if you really need this information). For convenience we should add these IDs as enumerated constants. Add the following code to the end of the `public` section of the `MainComponent.h` file:

```
...
    enum MenuIDs {
        LabelClear = 1000,
        SliderMin,
        SliderMax
    };
...
```

Notice that the first item is given a value of 1000; this is because an ID value of 0 (which is otherwise the default) is not valid as an ID. We need to store the `MenuBarComponent` object too. Add the code as highlighted below:

```
...
private:
    Label label;
    Slider slider;
    MenuBarComponent menuBar;
    ApplicationProperties appProperties;
};
```

In the `MainComponent.cpp` file, update the constructor for the `MainContentComponent` class as highlighted in the following:

```
...
MainContentComponent::MainContentComponent()
: menuBar (this)
{
    addAndMakeVisible (&menuBar);

    label.setEditable (true);
...

```

Here we pass the `this` pointer to the `MenuBarComponent` object in the initializer list. This is to tell the `MenuBarComponent` object which `MenuBarModel` to use. Update the `resized()` function in the `MainComponent.cpp` file to position the components as follows:

```
void MainContentComponent::resized()
{
    menuBar.setBounds (0, 0, getWidth(), 20);
    label.setBounds (10, 30, getWidth() - 20, 20);
    slider.setBounds (10, 60, getWidth() - 20, 20);
}
```

This positions the menu bar at the top of the window, filling the whole width of the window. Now we will add the menu bar functionality by implementing the virtual functions from the `MenuBarModel` class. Add the following code to the `MainComponent.cpp` file:

```
StringArray MainContentComponent::getMenuBarNames()
{
    const char* menuNames[] = { "Label", "Slider", 0 };
    return StringArray (menuNames);
}
```

This creates the top-level menu names by returning a `StringArray` object containing the names. Here we will have two menus, one to control the label, and the other to control the slider. Next, add the following code to the `MainComponent.cpp` file:

```
PopupMenu MainContentComponent::getMenuForIndex
(int index, const String& name)
{
    PopupMenu menu;

    if (name == "Label")
    {
        menu.addItem (LabelClear, "Clear");
    } else if (name == "Slider") {
        menu.addItem (SliderMin, "Set to minimum");
        menu.addItem (SliderMax, "Set to maximum");
    }

    return menu;
}
```

This checks which menu should be populated by inspecting the menu name. The **Label** menu will be filled with a single item that will be used to clear the label contents. The **Slider** menu will be filled with two items: one to set the slider to its minimum value, and one to set the slider to its maximum value. Notice that this is one place where we use the enumerated constants created earlier. Finally, add the following code to the `MainComponent.cpp` file:

```
void MainContentComponent::menuItemSelected (int menuID,
                                             int index)
{
    switch (menuID) {
        case LabelClear:
            label.setText (String::empty, dontSendNotification);
            break;
        case SliderMin:
            slider.setValue (slider.getMinimum());
            break;
        case SliderMax:
            slider.setValue (slider.getMaximum());
            break;
    }
}
```

Here we check which menu ID was selected by the user and act accordingly. Build and run the application to check this functionality. An additional example project `Chapter05_04b` is provided in the code bundle that illustrates how to modify this example to use the native menu bar on the Mac OS X platform. A more sophisticated technique for implementing menu bars is to use the JUCE `ApplicationCommandManager` class, which is used by the JUCE Demo application and the Introjucer application code to present its menus, issue commands from buttons, and so on. Refer to the JUCE documentation for this class for a complete guide.

## Summary

This chapter has introduced a range of additional useful utilities for application development in JUCE. This included using the `ValueTree` class and related classes for structuring and storing application data and properties, and adding undo management. This chapter also looked at multithreading support in JUCE, and introduced one final user interface component for adding menu bar controls to JUCE applications. These really are the tip of the iceberg. It is rare to find a JUCE class that is difficult to integrate into your own code. You are encouraged to explore the JUCE documentation to find further classes that will support your development. The JUCE code and classes introduced in this book should have given you an insight into the idioms of JUCE code. This should make discovering and using new JUCE classes relatively straightforward.



# Index

## A

**addAndMakeVisible() function** 27

**addListener() function** 33

**Affero General Public License** 8

**application properties**

  getBoolValue() function 132

  getDoubleValue() function 132

  getIntValue() function 132

  getUserSettings() function 132

  getValue() function 132

  setStorageParameters() function 132

  storing 130, 132

**Array class**

  about 74, 75

  banks of controls 82-85

  components array, creating 77-79

  files, finding in directory 76

  OwnedArray class, using 80-82

  strings, tokenizing 77

**AudioDeviceManager object** 104

**audio file playing support**

  adding 103-109

**audio files**

  controlling, GUI used 101, 102

  playing 100

**AudioTransportSource object** 108

## B

**Binary Builder tool**

  about 110

  image file, embedding using Introjucer

    application 110, 111

**binary files**

  reading 92, 93

  writing 92, 93

**broadcasters** 31

**buttonClicked() function** 31, 38, 99, 129

## C

**changeState() function** 108

**child components**

  adding 26, 28

**child node** 120

**colors**

  setting, look and feel used 41, 43

  specifying 38, 40

**complex component arrangements**

  configuring 52-55

**component color IDs** 40

**component types**

  MainComponent.cpp 37

  MainComponent.h 37

  using 37, 55

## D

**data entry**

  filtering 35, 36

**data structures**

  about 57

  numerical types 58

  text strings 59

**directory structures**

  employing 85-88

**Doxygen application**

  URL 21



## **drawing operations**

using 43-47

## **dynamically typed objects**

about 114

hierarchical data, structuring 116-121

Value class, using 114, 116

## **E**

### **examples, JUCE**

Codex Digital 21

Korg 21

M-Audio 21

Max 21

TC Group 21

Tracktion music 21

## **F**

**FileChooser::browseForFileToOpen()**  
function 100

**FileChooser::browseForFileToSave()**  
function 100

**FileChooser** class 55

**File::findChildFiles()** function 76

**FileNameComponent** class 55

### **file paths**

directory structures, navigating 72, 73

file information, obtaining 70

special directory locations, accessing 69

special locations 71

specifying 68

**FileTreeComponent** class 55

**focus ordering** 121

## **G**

**getHeight()** function 27

**getMenuBarNames()** function 134

**getMenuForIndex()** function 134

**getToggleState()** function 38

**getWidth()** function 27

**GNU General Public License** 8

### **GUI**

creating, for controlling audio file  
playing 101, 102

### **GUI, JUCE**

creating 23

## **H**

### **hierarchical data, dynamically type objects**

structuring 116-120

## **I**

**ImageComponent::setImagePlacement()**  
function 96

### **image data**

manipulating 97

**ImageFileFormat::loadFrom()** function 96

### **image files**

reading 94

writing 94

**initialise()** function 41

### **Integrated Development**

**Environment (IDE)** 8

**Internet Systems Consortium**  
(ISC) license 8

### **Introjuicer application**

about 13

building 13

building, on Mac OS X 14

building, on Windows 14

examining 14, 16, 17

JUCE project, creating with 18, 19

MainComponent.cpp file 19

MainComponent.h file 19

Main.cpp file 19

master-detail interface 15

tasks 13

## **J**

### **JUCE**

about 7

application properties, storing 130, 132

Array class 74

data structures 57

documentation 21

dynamically typed objects, using 114

examples 21

file paths, specifying 68

installing, for Mac OS X 7

installing, for Windows 7

memory management classes 88

menu bar controls, adding 133-136

- multiple threads, handling 127, 129
- source code, downloading 8
- Time class 62
- undo management, employing 121, 123
- URL, for documentation 21
- XML support, adding 124, 126
- JUCE Component class 24**
- JUCE coordinate system**
  - about 24
  - child components 24
  - single parent component 24
- juce\_core module 8**
- JUCE Demo application**
  - building 9
  - components 11
  - look and feel, customizing 12
  - overview 11
  - running, on Mac OS X 10
  - running, on Windows 9, 10
  - structure 9
- JUCE licensing**
  - reference link 8
- JUCE project**
  - creating, with Introjucer application 18-20

## L

- listeners 31**
- look and feel, JUCE Demo application**
  - customizing 12

## M

- Mac OS X**
  - Introjucer application, running on 14
  - JUCE Demo application, running on 10
  - JUCE, installing 8
- main() function 60**
- media files**
  - about 89
  - audio files, playing 100
  - binary files, reading 92, 93
  - binary files, writing 92, 93
  - image data, manipulating 97-100
  - image files, reading 94, 96
  - image files, writing 94, 96
  - simple input stream, using 89

- simple output stream, using 89
- text files, reading 90
- text files, writing 90
- menu bar controls**
  - adding 133-136
- menu bar model 133**
- menuItemSelected() function 134**
- Microsoft Visual Studio**
  - about 8
  - downloading 8
- Microsoft Visual Studio IDE 8**
- mouse activity**
  - intercepting 48-51
- mouseDown() function 48**
- mouseDrag() function 48**
- mouseEnter() function 48**
- mouseExit() function 48**
- mouseMove() function 48**
- mouseUp() function 48**
- multiple threads**
  - handling 127-129

## N

- numerical types 58**

## O

- observer pattern 31**
- OwnedArray class**
  - using 80-82

## P

- paint() function 44, 45**
- PopupMenu class 55**
- pure virtual function 31**

## R

- readFile() function 90**
- readFloatBigEndian() function 93**
- readFloat() function 93**
- readIntBigEndian() function 93**
- readInt() function 93**
- reference counted objects 59**
- resized() function 27**

## S

**setBounds()** function 27  
**setBoundsRelative()** function 28  
**setResizable()** function 28  
**setSize()** function 26, 27  
**slider**  
    text box, removing 31  
**sliderValueChanged()** function 31, 99

## T

**TabbedComponent** class 55  
**TableListBox** class 55  
**text box**  
    removing, from slider 31  
**TextEditor** class 55  
**text files**  
    reading 90, 91  
    writing 90, 91  
**text strings**  
    log messages, posting to console 59  
    manipulating 59-62  
    specifying 59  
**Thread::run()** function 129  
**Thread::stopThread()** function 129  
**Time** class  
    about 62  
    time data, manipulating 64, 66  
    time information, displaying 63  
    time information, formatting 63  
    time, measuring 66, 67  
**Time::formatted()** function 64  
**Time::getCurrentTime()** function 62, 66  
**Time::getMillisecondCounter()**  
    function 63, 66  
**time information**  
    displaying 63  
    formatting 63  
**Time\*\*toString()** function 63  
**ToggleButton** object 38  
**toggle-type** button 37  
**ToolBarButton** class 55  
**TreeView** class 55

## U

**undo management**  
    employing 121-123  
**user interaction**  
    responding to 30  
**user interfaces, JUCE**  
    building 24  
    buttons, creating 24  
    child components, adding 26-29  
    complex component arrangements,  
        configuring 52-54  
    components, creating 24  
    sliders, creating 24  
    user interaction, responding to 30

## V

**Value** class  
    using 114, 116

## W

**Windows**  
    Introjucer application, running on 14  
    JUCE Demo application, running on 9  
    JUCE, installing 8  
**writeFile()** function 90  
**writeFloatBigEndian()** function 93  
**writeFloat()** function 93  
**writeIntBigEndian()** function 93  
**writeInt()** function 93  
**writeToLog()** function 60

## X

**Xcode** IDE 8  
**XML support**  
    adding 124-126



## Thank you for buying Getting started with JUCE

### About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

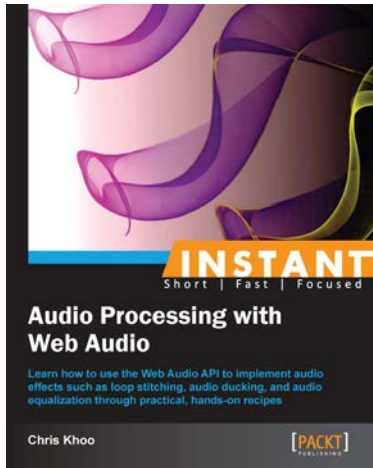
### About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



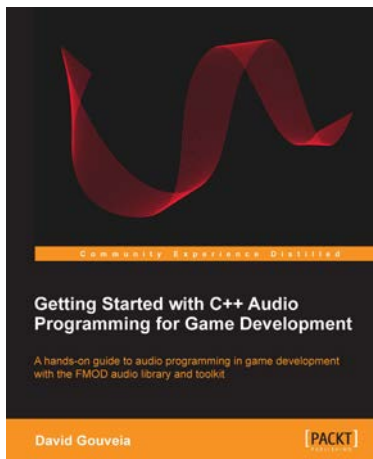
## Instant Audio Processing with Web Audio

ISBN: 978-1-782168-79-9

Paperback: 76 pages

Learn how to use the Web Audio API to implement audio effects such as loop stitching, audio ducking, and audio equalization through practical, hands-on recipes

1. Learn something new in an Instant!  
A short, fast, focused guide delivering immediate results
2. Implement audio stitching using Web Audio's built-in scheduling API
3. Learn Web Audio's scripting abilities by building a broadcast-style audio ducking effect



## Getting Started with C++ Audio Programming for Game Development

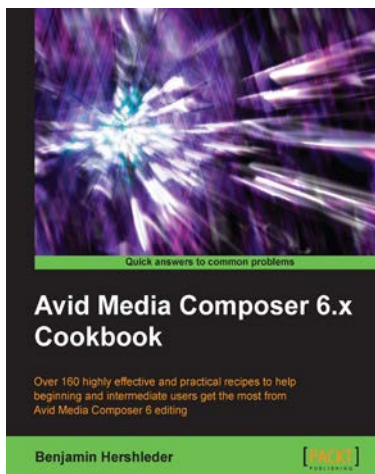
ISBN: 978-1-849699-09-9

Paperback: 116 pages

A hands-on guide to audio programming in game development with the FMOD audio library and toolkit

1. Add audio to your game using FMOD and wrap it in your own code
2. Understand the core concepts of audio programming and work with audio at different levels of abstraction
3. Work with a technology that is widely considered to be the industry standard in audio middleware

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



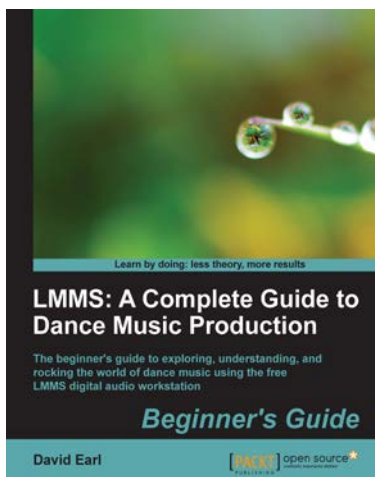
## Avid Media Composer 6.x Cookbook

ISBN: 978-1-849693-00-4

Paperback: 422 pages

Over 160 highly effective and practical recipes to help beginning and intermediate users get the most from Avid Media Composer 6 editing

1. Hands-on recipes in a step-by-step logical approach to quickly get started with Avid Media Composer and gain deeper understanding
2. Learn Avid Media Composer in a completely new way – gain intensive exposure with various editing options to develop your abilities, become even more creative, and acquaint yourself with various methods that you never thought were possible



## LMMS: A Complete Guide to Dance Music Production

ISBN: 978-1-849517-04-1

Paperback: 384 pages

The beginner's guide to exploring, understanding, and rocking the world of dance music using the free LMMS digital audio workstation

1. Create the dance music you wanted. An experienced guide shows you the ropes.
2. Learn from the best in dance music; its history, its people, and its genres.
3. Learn the art of making music: from the way you set up your equipment, to polishing up your final mix.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles