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.