Appendix C. Differences between FOX and FXRuby

The FXRuby API follows the FOX API very closely and for the most part, you should be able to use the standard FOX class documentation as a reference. In some cases, however, fundamental differences between Ruby and C++ necessitated slight changes in the API. For some other cases, FOX classes were enhanced to take advantage of Ruby language features (such as iterators). The purpose of this chapter is to identify some of the differences between the C++ and Ruby interfaces to FOX.

One difference that should be easy to cope with is the substitution of Ruby Strings for FXStrings. Any function that would normally expect an FXString input argument insteads takes a Ruby String. Similarly, functions that would return an FXString will instead return a Ruby string. For functions that would normally accept a `NULL` or empty string argument, just pass `nil` or an empty string ("").

Functions that expect arrays of objects

One common pattern in FOX member function argument lists is to expect a pointer to an array of values, followed by an integer indicating the number of values in the array. This of course isn't necessary in Ruby, where `Array` objects "know" their lengths. As a result, functions such as `FXWindow::acquireClipboard()`, whose C++ declaration looks like this:

```
FXbool acquireClipboard(const FXDragType *types, FXuint numTypes);
```

are called from Ruby code by passing in a single `Array` argument, e.g.

```
myWindow.acquireClipboard(typesArray)
```

Functions that return values by reference

Many FOX methods take advantage of the C++ language feature of returning values by reference. For example, the `getCursorPos()` member function for class `FXWindow` has the declaration:

```
FXint getCursorPos(FXint& x, FXint& y, FXint& buttons) const;
```

which indicates that the function takes references to three integers (x, y and buttons). To call this function from a C++ program, you'd write code like this:

```
FXint x, y;
FXuint buttons;

if (window->getCursorPosition(x, y, buttons))
```

```
  fprintf(stderr, "Current position is (%d, %d)\n", x, y);
```

Since this idiom doesn't translate well to Ruby, some functions' interfaces have been slightly modified. For example, the FXRuby implementation of `getCursorPos()` returns the three values as an `Array`, e.g.:

```
x, y, buttons = aWindow.getCursorPos()
```

The following table shows how these kinds of functions are implemented in FXRuby:

Instance Method	Return Value
`FXDial#range`	Returns a `Range` instance.
`FXDial#range=(aRange)`	Accepts a `Range` instance as its input.
`FXFontDialog#fontSelection`	Returns the `FXFontDesc` instance
`FXFontSelector#fontSelection`	Returns the `FXFontDesc` instance
`FXGLObject#bounds(range)`	Takes an `FXRange` instance as its input and returns a (possibly modified) `FXRange` instance.
`FXGLViewer#eyeToScreen(eye)`	Takes an array of eye coordinates (floats) as its input and returns the screen point coordinate as an array of integers [sx, sy]
`FXGLViewer#getBoreVector(sx, sy)`	Returns the endpoint and direction vector as an array of arrays [point, dir]
`FXGLViewer#light`	Returns a `FXLight` instance
`FXGLViewer#viewport`	Returns an `FXViewport` instance.
`FXPrinterDialog#printer`	Returns the `FXPrinter` instance
`FXScrollArea#position`	Returns the position as an array of integers [x, y]
`FXSlider#range`	Returns a `Range` instance.
`FXSlider#range=(aRange)`	Accepts a `Range` instance as its input.
`FXSpinner#range`	Returns a `Range` instance.
`FXSpinner#range=(aRange)`	Accepts a `Range` instance as its input.
`FXText#appendText(text, notify=false)`	Append text to the end of the buffer.
`FXText#appendStyledText(text, style=0, notify=false)`	Append styled text to the end of the buffer.
`FXText#extractText(pos, n)`	Extracts *n* characters from the buffer beginning at position *pos* and returns the result as a String.
`FXText#extractStyle(pos, n)`	Extracts *n* style characters from the buffer beginning at position *pos* and returns the result as a String.
`FXText#insertText(pos, text, notify=false)`	Insert *text* at position *pos* in the buffer.
`FXText#insertStyledText(pos, text, style=0, notify=false)`	Insert *text* at position *pos* in the buffer.
`FXText#replaceText(pos, m, text, notify=false)`	Replace *m* characters at *pos* by *text*.

`FXText#replaceStyledText(pos, m, text, style=0, notify=false)`	Replace *m* characters at *pos* by *text*.
`FXText#setDelimiters(delimiters)`	Change delimiters of words (*delimiters* is a string).
`FXText#getDelimiters()`	Return word delimiters as a string.
`FXWindow#cursorPosition`	Returns an array of integers [x, y, buttons]
`FXWindow#translateCoordinatesFrom(window, x, y)`	Returns the translated coordinates as an array [x, y]
`FXWindow#translateCoordinatesTo(window, x, y)`	Returns the translated coordinates as an array [x, y]

Iterators

Several classes have been extended with an `each` method to provide Ruby-style iterators. These classes include `FXComboBox`, `FXGLGroup`, `FXHeader`, `FXIconList`, `FXList`, `FXListBox`, `FXTreeItem`, `FXTreeList` and `FXTreeListBox`. These classes also mix-in Ruby's `Enumerable` module so that you can take full advantage of the iterators.

The block parameters passed to your code block vary depending on the class. For example, iterating over an `FXList` instance yields `FXListItem` parameters:

```
aList.each { |aListItem|
    puts "text for this item = #{aListItem.getText()}"
}
```

whereas iterating over an `FXComboBox` instance yields two parameters, the item text (a string) and the item data:

```
aComboBox.each { |itemText, itemData|
    puts "text for this item = #{itemText}"
}
```

The following table shows the block parameters for each of these classes' iterators:

Class	Block Parameters
`FXComboBox`	the item text (a string) and user data
`FXGLGroup`	an `FXGLObject` instance
`FXHeader`	an `FXHeaderItem` instance
`FXIconList`	an `FXIconItem` instance
`FXList`	an `FXListItem` instance
`FXListBox`	the item text (a string), icon (an `FXIcon` instance) and user data
`FXTreeItem`	an `FXTreeItem` instance
`FXTreeList`	an `FXTreeItem` instance
`FXTreeListBox`	an `FXTreeItem` instance

Attribute Accessors

FOX strictly handles access to all object attributes through member functions, e.g. `setBackgroundColor` and `getBackgroundColor` or `setText` and `getText`. FXRuby exposes all of these functions but also provides aliases that look more like regular Ruby attribute accessors. The names for these accessors are based on the FOX method names; for example, `setBackgroundColor` and `getBackgroundColor` are aliased to `backgroundColor=` and `backgroundColor`, respectively.

In many cases these aliases allow you to write more compact and legible code. For example, consider this code snippet:

```
aLabel.setText(aLabel.getText() + " (modified)")
```

Now consider a different code snippet, using the aliased accessor method names:

```
aLabel.text += " (modified)"
```

While these two are functionally equivalent, the latter is a bit easier to read and understand at first glance.

Message Passing

FOX message maps are implemented as static C++ class members. With FXRuby, you just associate messages with message handlers in the class `initialize` method using the `FXMAPFUNC()`, `FXMAPTYPE()`, `FXMAPTYPES()` or `FXMAPFUNCS()` methods. See almost any of the example programs for examples of how this is done.

As in C++ FOX, the last argument passed to your message handler functions contains message-specific data. For instance, all `SEL_PAINT` messages pass an `FXEvent` object through this argument to give you some information about the size of the exposed rectangle. On the other hand, a `SEL_COMMAND` message from an `FXHeader` object passes the index of the selected header item through this argument. Instead of guessing what's in this last argument, your best bet is to instead invoke a member function on the sending object to find out what you need, instead of relying on the data passed through this pointer. For example, if you get a `SEL_COMMAND` message from an `FXColorWell` object, the data passed through that last argument is supposed to be the new RGB color value. Instead of trying to interpret the argument's contents, just turn around and call the color well's `getRGBA()` member function to retrieve its color. Similarly, if you get a `SEL_COMMAND` message from a tree list, call its `getCurrentItem()` method to find out which item was selected.

Catching Operating System Signals

The `FXApp#addSignal` and `FXApp#removeSignal` methods have been enhanced to accept either a string or integer as their first argument. If it's a string (e.g. "SIGINT" or just "INT") the code will determine the corresponding signal number for you (similar to the standard Ruby library's `Process.kill` module method). For examples of how to use this, see the *datatarget.rb* or *imageviewer.rb* example programs.

Support for Multithreaded Applications

There is some support for multithreaded FXRuby applications, but it's not wonderful. The current implementation does what is also done in Ruby/GTK; it turns over some idle processing time to the Ruby thread scheduler to let other threads do their thing. As I learn more about Ruby's threading implementation I may try something different, but this seems to work OK for now. For a simple example, see the *groupbox.rb* example program, in which the clock label that appears in the lower right-hand corner is continuously updated (by a separate thread).

If you suspect that FXRuby's threads support is interfering with your application's performance, you may want to try tweaking the amount of time that the main application thread "sleeps" during idle processing; do this by setting the `FXApp` object's *`sleepTime`* attribute. The default value for *`FXApp#sleepTime`* is 100 milliseconds. You can also disable the threads support completely by calling `FXApp#threadsEnabled=false` (and subsequently re-enable it with `FXApp#threadsEnabled=true`).

Keyword-Style Arguments

FXRuby 1.6.5 introduced preliminary, experimental support for using keyword-style arguments in FXRuby method calls. The current implementation of this feature only works for class constructors (i.e. the "new" class methods), but the intent is to gradually extend this feature so that it covers all FXRuby methods.

What this means in practice is that for calls to methods that have one or more optional arguments, you can replace all of the optional arguments with a hash that sets only the values for which the default isn't appropriate. So, for example, consider a typical call to `FXMainWindow.new`:

```
main = FXMainWindow.new(app, "Title Goes Here", nil, nil, DECOR_ALL, 0, 0, 800, 600)
```

In this case, the programmer wants to set the initial window width and height to 800 and 600. In order to do that (with the current release of FXRuby), however, she's required to fill in all of the optional arguments in between the window title string and the width and height values. As anyone who's worked with FXRuby for any amount of time will tell you, it's easy to accidentally leave out one of those intermediate arguments and it can be difficult to figure out what's wrong afterwards.

Now consider how this method call could be written using the new keyword arguments support. First, the programmer would need to require the keyword arguments library:

```
require 'fox16/kwargs'
```

Then she would look up the API documentation for the `FXMainWindow#initialize` method (e.g. here) and see that the names of the width and height arguments are, in fact, "width" and "height". Armed with that information, she would be able to rewrite the previous code as:

```
main = FXMainWindow.new(app, "Title Goes Here", :width => 800, :height => 600)
```

Here, the programmer has omitted the intermediate optional arguments (thus accepting their default values) and specified only the width and height values. This code is obviously a lot more readable and maintainable.

It is important to observe the difference between required and optional arguments when using this feature. If the API documentation for a particular method doesn't indicate that an argument has a default value, then it is by definition not an optional argument. So one could *not* write the example as, e.g.

```
main = FXMainWindow.new(app, :width => 800, :title => "Title Goes Here", :height => 600)
```

This example is incorrect because the title argument is required, and it must be the second argument in the call. Obviously, this means that the optional arguments in a method call (if they're specified) will always follow all of the required arguments.

Finally, note that the keyword arguments feature has been implemented so that it's backwards-compatible with the original positional arguments scheme (or it's intended to be, at any rate). What that means is that you can immediately start making use of this feature in your existing code, even if you don't have time to update all of the method calls to use keyword arguments.

Debugging Tricks

As a debugging tool, you can optionally catch exceptions raised in message handlers. To turn on this feature, call the `setIgnoreExceptions(true)` module method. When this is enabled, any exceptions raised in message handler functions will cause a standard stack trace to be dumped to the standard output, but then your application will, for better or worse, proceed normally. Thanks to Ted Meng for this suggestion.

Prev	Up	Next
Appendix B. Using Scintilla with FXRuby	Home	Appendix D. The FXRuby Standard Library