Extracted from:

FXRuby

Create Lean and Mean GUIs with Ruby

The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before we normally would. That way you'll be able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned. The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos and other weirdness. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long lines with little black rectangles, incorrect hyphenations, and all the other ugly things that you wouldn't expect to see in a finished book. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Throughout this process you'll be able to download updated PDFs from your account on http://pragprog.com. When the book is finally ready, you'll get the final version (and subsequent updates) from the same address. In the meantime, we'd appreciate you sending us your feedback on this book at http://books.pragprog.com/titles/fxruby/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

► **Andy Hunt**

Chapter 8

Sorting Data with List and Table Widgets

The simple widgets that we learned about in the previous chapter primarily deal with a single value (if they have any real "value" associated with them at all). FXRuby also provides a number of more complicated widgets for dealing with collections of values. Figure 8.1, on the next page lists the widgets that we'll be looking at in this chapter, along with brief descriptions of when you'd want to consider using them in your applications. We'll begin by looking at the FXList.

8.1 Displaying Simple Lists with FXList

The FXList widget displays a list of items, where each item has an associated text string and an optional icon. If the list contains more items than it can display, it will grow a vertical scrollbar to allow you to scroll up or down in the list.

By default, an FXList is empty. You can add items to the end of a list using the appendItem() method.

Download listexample.rb

```
groceries = FXList.new(self,
  :opts => LIST_NORMAL|LIST_EXTENDEDSELECT|LAYOUT_FILL)
groceries.appendItem("Milk")
groceries.appendItem("Eggs")
groceries.appendItem("Bacon (Chunky)")
```

You can of course also prepend an item to the beginning of the list, insert an item at a specific position in the list, or remove an item from

Widget Class	What's it for?
FXList	Use FXList to display an always-visible, flat list of items and allow the user to select one or more items from it.
FXListBox	Use FXListBox to display a drop-down, flat list of items and allow the user to select a single item from it.
FXComboBox	Use FXComboBox to display a drop-down, flat list of items and allow the user to select a single item from it. Unlike FXListBox, the FXComboBox is editable.
FXTreeList	Use FXTreeList to display a list of hierarchically structured items and allow the user to select one or more items from it.
FXTable	Use FXTable to display a collection of items in tabular form and allow the user to select one or more items from it.

Figure 8.1: List Widgets

the list (using the prependItem(), insertItem() or removeItem() method, respectively).

Download listexample.rb

```
groceries.prependItem("Bread")
groceries.insertItem(2, "Peanut Butter")
groceries.removeItem(3)
```

Making Selections in Lists

FXRuby maintains several attributes having to do with the current selection in a list. The *current item* is simply the last list item that you clicked on, and it's the item that currently has the keyboard focus. If there is no current item, the currentItem for a list is -1; otherwise, it's the integer index of the current item. When the current item changes, the FXList sends both a SEL_CHANGED and SEL_COMMAND message to the list widget's target.

```
groceries.connect(SEL_COMMAND) do |sender, sel, index|
  puts "The new current item is #{sender.currentItem}"
end
```

The list sends a number of other interesting messages to its target when, for example, the user double-clicks on a list item. For a com-

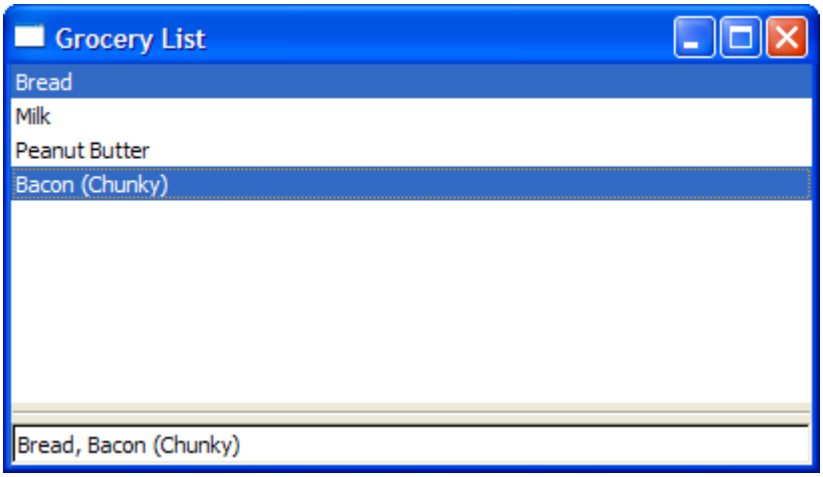

Figure 8.2: FXList in Action

plete listing of all the messages that FXList sends to its target, check the API documentation.

The selection mode for an FXList sets the policy for how many items can be selected at the same time, and how you go about changing the selection. One selection mode that you'll use often is the LIST_BROWSESELECT mode. In this mode, there's always exactly one list item selected, and it's the last one you clicked on. The other commonly used list selection mode is LIST_EXTENDEDSELECT mode. In this mode, any number of items can be selected. Control-clicking an item toggles its selected state, and holding down the Shift key while clicking on items will extend the current selection to include all of the intermediate items.

TheFXListalso provides the less-frequently usedLIST_SINGLESELECT,LIST_AUTOS modes.

So Which Items Are Selected?

When the list is configured in either the LIST_SINGLESELECT, LIST_BROWSESELECT or LIST_AUTOSELECT mode, you can safely assume that the currentItem is the currently selected item. When the list is configured in either LIST_EXTENDEDSELECT or LIST_MULTIPLESELECT mode, however, you must check each list item individually to find out whether it's selected. One way to do this is to iterate over all of the item indices.

```
selected_indices = []
0.upto(list.numItems-1) do |index|
  selected_indices << index if list.itemSelected?(index)
```

```
end
```

A different approach is to iterate over the FXListItem instances themselves, testing their selected?() states.

Download listexample.rb

```
selected_items = []
groceries.each { |item| selected_items << item if item.selected? }
```

As you might expect, the FXList and FXListItem classes provide a number of additional methods having to do with the behavior and appearance of a list. For all the gory details, see the API documentation for these classes.

Depending on the number of items in the list, and the available "real estate" in your user interface, an FXList might not be the best choice for displaying a collection of data. If you need to display a long list of items, but only have a small amount of space to work with, a combo box or list box might work better. We'll take a look at those widgets next.

8.2 Good Things Come in Small Packages: FXComboBox and FXListBox

The FXComboBox and FXListBox widgets are both variations on the FXList widget. Both of these widgets look like a combination of an FXTextField and an FXArrowButton. When you click the arrow button, the text field expands to display the entire list of items. After you select an item from the list, the list "pops" back down to assume its original appearance. Like FXList, they can both be used to display a flat list of items from which the user can select an item. Unlike FXList, they only allow you to select one item at a time from the list.

There are no hard and fast rules about when it's preferable to use a regular FXList as opposed to an FXComboBox or FXListBox. Obviously, if you need for the user to be able to pick more than one item, you'd want to go with the FXList. On the other hand, if a single selection is appropriate and if you don't have enough room in the user interface to display a list, a combo box or list box is a nice, compact way to hide the list's contents away when they aren't needed.

My personal philosophy is that if a list is going to contain more than a handful of items, I'll use a combo box or list box instead of a plain old list. The differences between combo boxes and list boxes are subtle, however, and depending on how you use them they're pretty inter-

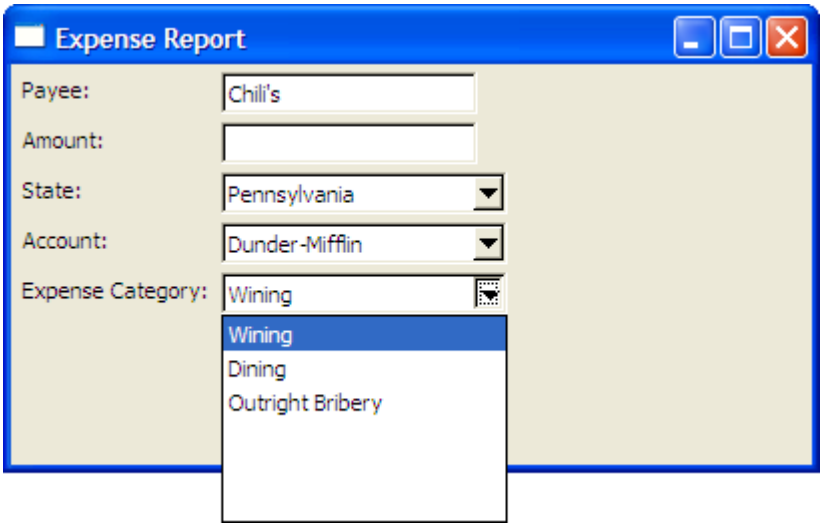

Figure 8.3: Separated at Birth? FXComboBox and FXListBox

changeable. I mean, can *you* tell the difference between the two in Figure 8.3? Neither can I. Basically, if all you need is to be able to select an item from a list, you should use the FXListBox. If you'd like to be able to type in a text string as an alternative to the existing list items, and even see that item added to the list of items, you should use FXComboBox.

Like FXList, both of these widgets provide prependItem(), appendItem(), insertItem() and removeItem() methods for altering the contents of the list.

Download comboboxexample.rb

```
states = FXListBox.new(matrix,
  :opts => LISTBOX_NORMAL|FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X)
$state_names.each { |name| states.appendItem(name) }
```

One item can be selected at any time, and the currentItem attribute indicates the index of that item (or -1 if there is no current item).

Since the FXComboBox can be edited, there are a few additional issues that we need to address for that widget. One issue has to do with whether text that the user types into the combo box's text field should be added to the list of items or not. By default, the combo box uses the COMBOBOX_NO_REPLACE option, which means that the list's contents remain the same regardless of what the user types into the text field.

Download comboboxexample.rb

```
accounts = FXComboBox.new(matrix, 20,
  :opts => COMBOBOX_NO_REPLACE|FRAME_SUNKEN|FRAME_THICK|LAYOUT_FILL_X)
```

When you're using an editable FXComboBox, you can't necessary depend on the currentItem to lead you to the user's input, since they may have typed some new text into the text field. For that reason, you should instead inspect the value of the text attribute to determine the combo box's current value.

Download comboboxexample.rb

```
accounts.connect(SEL_COMMAND) do |sender, sel, index|
  assign_expense_account(sender.text)
end
```

If you'd like for the strings that the user types into the text field to be added to the combo box's list, you have several choices as to where those new items are placed in the list.

- Use the COMBOBOX_INSERT_FIRST option to insert the new item at the beginning of the list.
- Use the COMBOBOX_INSERT_LAST option to insert the new item at the end of the list.
- Use the COMBOBOX_INSERT_BEFORE option to insert the new item before the current item.
- Use the COMBOBOX_INSERT_AFTER option to insert the new item after the current item.

In my experience, the COMBOBOX_INSERT_BEFORE and COMBOBOX_INSERT_AFTER options are a bit confusing, from a user's perspective, and I usually just stick with the COMBOBOX_INSERT_FIRST option. Note that the FXComboBox doesn't have a built-in option to automatically maintain the sort order of the items, but you can work around this by calling sortItems() on the combo box during the SEL_COMMAND handler.

Download comboboxexample.rb

```
categories.connect(SEL_COMMAND) do |sender, sel, index|
  assign_expense_category(sender.text)
  sender.sortItems
end
```

The call to sortItems() won't disturb the text entered in the text field, but if you click the arrow button to pop the list pane down, you will see that the newly added item appears at the correct position in the sorted list.

The widgets that we've looked at so far in this chapter all deal with flat lists of items. FXRuby also provides support for dealing with hierarchically structured data by way of the FXTreeList widget, and we'll discuss it next.

8.3 Branching Out with Tree Lists

The FXTreeList widget is so named because you can imagine the data that it manages as tree-like, starting from a root and reaching out in various directions, with branches leading to other branches. Unlike the FXList, FXComboBox and FXListBox, which all deal with flat lists of things, the FXTreeList is designed for use with hierarchically structured data. Although we use the word "tree" to describe this list's data and appearance, you should note that it's not exactly like the classic tree data structure that you may have studied in your computer science classes. One especially confusing point is that the standard documentation for the FXTreeList class uses the term "root" item to refer to any one of the top-most visible items in the tree. From a strict computer-science point of view, the actual root of the tree never appears on screen, and we can only refer to it indirectly using the FXTreeList API.

Once you get used to the terminology that FXRuby uses to talk about the FXTreeList, however, you'll find that it's easy to use in practice. You can modify the content of the tree list using the familiar prependItem(), insertItem(), appendItem() and removeItem() methods, although the calling conventions are slightly different due to the hierarchical nature of the list. The first argument for the prependItem() and appendItem() methods is a reference to the parent item for the item that you're adding. If it's a top-level item, pass in nil as the first argument.

Download treelistexample.rb

```
treelist = FXTreeList.new(treelist_frame,
  :opts => TREELIST_NORMAL|TREELIST_SHOWS_LINES| \
           TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES|LAYOUT_FILL)
artist_1    = treelist.appendItem(nil, "Alison Kraus")
album_1_2   = treelist.appendItem(artist_1, "Forget About It")
track_1_2_3 = treelist.appendItem(album_1_2, "Ghost in this House")
track_1_2_2 = treelist.prependItem(album_1_2, "Maybe")
track_1_2_1 = treelist.insertItem(track_1_2_2, album_1_2, "Stay")
album_1_1   = treelist.prependItem(artist_1, "Every Time You Say Goodbye")
```

There are three options that you can use to control how the connections between parent and child items in the tree list are displayed. If the TREELIST_SHOWS_LINES option is selected, the tree list will draw a faint dotted

Figure 8.4: A Sample FXTreeList

line from a parent item to each of its child items. If TREELIST_SHOWS_BOXES is selected, the tree list will display a small box to the left of any tree item that has one or more child items; if that tree item is expanded, the box will contain a dash, and if the tree item is collapsed, it will contain a plus sign. Now, for some reason, the TREELIST_SHOWS_BOXES option only applies to items nested somewhere below the top-level items. If you also want to see the boxes next to top-level items (and remembering that FOX calls these the "root-level" items), you must also pass in the TREELIST_ROOT_BOXES option. Note that the TREELIST_ROOT_BOXES option has no effect unless TREELIST_SHOWS_BOXES is also enabled.

Having said all that, I usually pass in all three options, as shown in the sample code. I've never found a good reason to omit any of them. Figure 8.4 will give you an idea of what the tree list looks like in this case.

Keeping Track of the Selection

FXTreeList supports the same kinds of selection modes that FXList does, and they work in the same ways, so the things that you've already learned about them apply here as well. The currentItem attribute still tells you the last item that was clicked, although in this case it's a reference to an FXTreeItem object instead of an integer index.

Determining which items are selected in a tree list can be tricky, however, when the selection mode allows for multiple selected items. The most straightforward way to do this, in my experience, is to track the selected items in an Array (or some other container) and then use the SEL_SELECTED and SEL_DESELECTED messages from the FXTreeList to update the array.

Download treelistexample.rb

```
selected_items = []
treelist.connect(SEL_SELECTED) do |sender, sel, item|
  selected_items << item unless selected_items.include? item
end
treelist.connect(SEL_DESELECTED) do |sender, sel, item|
  selected_items.delete(item)
end
```

This technique works well for any size tree list because it's inexpensive, computationally speaking. If you know that the tree list isn't going to hold all that many items, however, you may find that simply traversing the tree every time the current item changes, and recording which items are selected, is fast enough for your purposes. Just catch the SEL_COMMAND message from the FXTreeList.

Download treelistexample.rb

```
treelist.connect(SEL_COMMAND) do |sender, sel, current|
  selected_items = []
  treelist.each { |child| add_selected_items(child, selected_items) }
end
```

Here's the add_selected_items(), which traverses the tree in a recursive fashion to see which items are selected.

Download treelistexample.rb

```
def add_selected_items(item, selected_items)
  selected_items << item if item.selected?
  item.each { |child| add_selected_items(child, selected_items) }
end
```

Now before we end this chapter, a super-secret bonus trick about associating a right-click popup menu with an FXTreeList.

Creating Context Menus for Tree Items

Users have gotten used to the idea of being able to right-click on an object in the user interface to display a context-sensitive popup menu for that object. You can do this with almost any kind of object in FXRuby, but it sure seems to come up a lot when developers decide to add an

FXTreeList to their application. For that reason, I'm going to treat you to a little recipe for how to add one of these right-click popup menus to a tree list, bearing in mind that a very similar technique could be applied to other widgets. I'm going to skim over the details about the different parts of the menu itself, but we'll cover that in depth later, in Chapter 12, *Working with Menus and Toolbars*, on page 154.

The first step is to catch the SEL_RIGHTBUTTONRELEASE message that the FXTreeList forwards to its target. You could instead trigger the popup on SEL_RIGHTBUTTONPRESS, but I think it feels more natural to do it when the button is released. Use the window coordinates reported in the event data to determine which tree item (if any) was hit.

Download treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
  item = sender.getItemAt(event.win_x, event.win_y)
  unless item.nil?
    # ...
  end
end
```

The getItemAt() method will return nil if there is no tree item at the specified coordinates. Otherwise, it will return a reference to that FXTreeItem. The next step is to construct an FXMenuPane and add one or more menu commands to it.

Download treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
  item = sender.getItemAt(event.win_x, event.win_y)
  unless item.nil?
    FXMenuPane.new(self) do |menu_pane|
      play = FXMenuCommand.new(menu_pane, "Play Song")
      play.connect(SEL_COMMAND) { play_song_for(item) }
      info = FXMenuCommand.new(menu_pane, "Get Info")
      info.connect(SEL_COMMAND) { display_info_for(item) }
      # ...
    end
  end
end
```

Finally, create the menu pane, call popup() on it to display it onscreen, and then start a nested run loop focused on that menu pane.

Download treelistexample.rb

```
treelist.connect(SEL_RIGHTBUTTONRELEASE) do |sender, sel, event|
  item = sender.getItemAt(event.win_x, event.win_y)
  unless item.nil?
    FXMenuPane.new(self) do |menu_pane|
```

Figure 8.5: Adding a Context Menu for the Tree List

```
      play = FXMenuCommand.new(menu_pane, "Play Song")
      play.connect(SEL_COMMAND) { play_song_for(item) }
      info = FXMenuCommand.new(menu_pane, "Get Info")
      info.connect(SEL_COMMAND) { display_info_for(item) }
      menu_pane.create
      menu_pane.popup(nil, event.root_x, event.root_y)
      app.runModalWhileShown(menu_pane)
    end
  end
end
```

As soon as the user clicks on one of the menu commands, or clicks outside the popup menu, the menu pane will be hidden and the application will fall back out of the event loop started by the call to runModalWhileShown(). Figure 8.5 shows what the popup menu looks like when I right-click on one of the songs in the list. This is an easy bit of code to add to an application, and when used properly it can really enhance the program's usability.

So now we have options for dealing with both flat lists of data as well as nested lists of data. Next up, we're going to take a look at one more of the widgets that FXRuby provides for dealing with collections of data, and that's the FXTable widget.

8.4 Table Widget

The FXTable widget is one of the more complicated widgets in the FOX toolkit, and it's one that has evolved pretty significantly since it was originally introduced. Newcomers sometimes confuse the FXTable widget with the FXMatrix layout manager, which you can use to lay out a bunch of widgets in rows and columns.[1] The FXTable*does* lay out its contents in rows and columns, but it's not a layout manager per se; in some other toolkits, you may have heard this kind of a widget referred to as a "grid" widget or "spreadsheet" widget.

Storing Data in a Table

Our study of FXTable begins with a look at how to create a table and add some data to it. In this section we're going to learn a little bit about how the table actually manages its data internally. We'll see that the table's sparse storage scheme makes it very efficient, and we'll also learn how to define items that can span multiple table cells.

Like the list widgets we looked at in the last chapter, tables are empty by default. The most efficient way to fill up a table is to use the setTableSize() method.

Download tableexample1.rb

```
table = FXTable.new(self, :opts => LAYOUT_FILL)
table.setTableSize(10, 10)
```

An important thing to recognize about setTableSize(), and all of the methods that alter the table's size, is that FXTable makes a distinction between empty cells and those that have some content (or data) associated with them. Both kinds of cells take up space onscreen when the table is drawn, but internally, FXTable only allocates storage (in the form of FXTableItem objects) for those cells that actually have content. This makes the table very efficient in terms of memory use, and it means that you can store pretty large tables with very little penalty.

You should also understand that the setTableSize() method is a destructive method. Whether you're initializing the table size, or simply resizing it to make it bigger or smaller, setTableSize() begins by destroying all of the existing table items. So if your table already contains some data,

1. We used the FXMatrix layout manager while building the Picture Book application, and we'll discuss it in more detail in the (as yet) unwritten *sec.layout.matrix*.

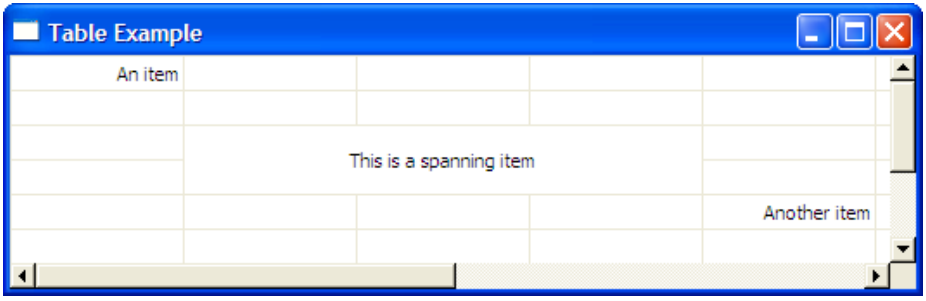

Figure 8.6: Table with a Spanning Item

and you just want to grow it by a few rows or columns, calling setTableSize() is not the way to do it. Instead, use some combination of the appendRows(), appendColumns(), insertRows() and insertColumns() methods.[2]

A *spanning item* is one that takes up more that one position in the table. You can create a spanning item by simply passing in the same item to setItem() for several adjacent rows and columns.

Download tableexample1.rb

```
table.setItemText(2, 1, "This is a spanning item")
table.setItemJustify(2, 1, FXTableItem::CENTER_X)
spanning_item = table.getItem(2, 1)
table.setItem(2, 2, spanning_item)
table.setItem(2, 3, spanning_item)
table.setItem(3, 1, spanning_item)
table.setItem(3, 2, spanning_item)
table.setItem(3, 3, spanning_item)
```

In this example, the item at position (2, 1) spans a 3x2 block of cells in the table. When this part of the table is drawn, none of the interior grid lines will be drawn. Figure 8.6 shows what the table looks like for this spanning item.

Modifying the Table Display Options

So far we've talked about how you can add to or modify the table data. An equally important topic is the display of that data, and more specif-

2. There aren't any methods to prepend rows or columns to a table, but you can use insertRows() or insertColumns(), passing in a value of zero for the starting row or column.

ically the amount of control that the user has over the table's appearance.

By default, both horizontal and vertical grid lines are displayed so that the borders of individual table cells are clearly delineated.[3] If you'd like to turn off the display of grid lines, set either or both of the horizontalGridShown and verticalGridShown attributes to false.

```
table.horizontalGridShown = false
```

All of the cells in a row have the same height, and all of the cells in a column have the same width. However, different rows can have different row heights, and different columns can have different column widths. By default, the user can't change those any of those sizes. You can always change the row heights and column widths programatically, using methods like setRowHeight() and setColumnWidth(), but to allow the user to interactively resize them, you must enable either the TABLE_ROW_SIZABLE flag, the TABLE_COL_SIZABLE flag, or both.

```
table.tableStyle |= TABLE_COL_SIZABLE
```

When one or both of these options is enabled, the user can click on the separator between two items in the row (or column) header and drag it from side to side to resize the neighboring rows (or columns).

Speaking of those row and column headers, you can also manipulate their contents to provide (for example) titles for the table columns.

Download tableexample2.rb

```
table.setColumnText(0, "Ruby 1.8.6")
table.setColumnText(1, "Ruby 1.9")
table.setColumnText(2, "JRuby")
table.setColumnText(3, "Rubinius")
```

If you want to turn off the display of the row header (a pretty common request), first change its mode to LAYOUT_FIX_WIDTH, and then set its width to zero pixels. You can do the same for the column header, by setting the columnHeaderMode to LAYOUT_FIX_HEIGHT and the columnHeaderHeight to zero.

Download tableexample1.rb

```
table.rowHeaderMode = LAYOUT_FIX_WIDTH
table.rowHeaderWidth = 0
table.columnHeaderMode = LAYOUT_FIX_HEIGHT
```

3. As we've already mentioned, the interior grid lines for spanning table items are never drawn.

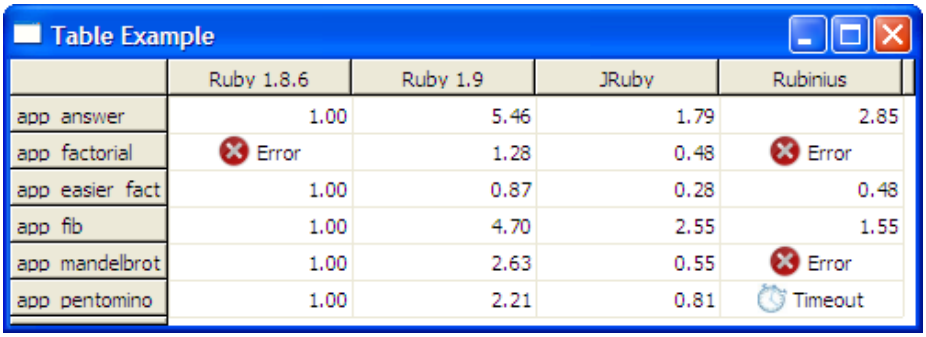

Figure 8.7: Table Items with Icons

```
table.columnHeaderHeight = 0
```

The table shown in Figure 8.6, on page 122 has both its row and column headers hidden.

You also have some degree of control over the display of individual table items. Each table item has an associated text string and icon. You can change these values using the setItemText() and setItemIcon() methods.

Download tableexample2.rb

```
table.setItemText(5, 3, "Timeout")
table.setItemIcon(5, 3, stopwatch_icon)
table.setItemJustify(5, 3, FXTableItem::CENTER_X)
table.setItemIconPosition(5, 3, FXTableItem::BEFORE)
```

The table shown in Figure 8.7 includes a number of items with the text right-justified (the default) as well as others with centered text and icons.

Finally, the user can edit the contents of a table cell by double-clicking on that cell, typing some new text, and pressing the Enter key. You can disable this feature by setting the editable attribute to false.

```
table.editable = false
```

So far we've been focusing on the display aspects of the FXTable: how to put data inside it, and how to change its appearance. Like the other widgets we've looked at in this chapter, however, the table is also useful as an input mechanism. To wrap up this section, we're going to take a look at how users can make selections in tables.

Managing the Table Selection

The table is somewhat less flexible than the list widgets in terms of its selection model. It supports only one selection mode, and in that mode you can select either a single cell, or a contiguous block of cells. You can't, for example, select one cell in the upper left corner, and another cell in the lower right corner, without also selecting all of the cells in between.

When you click in a cell to begin building up a selection, that cell becomes the *anchor cell*. The anchorRow and anchorColumn attributes for the table contain the row and column indices of the anchor item, assuming that there is one. If you then hold down the Shift key and click somewhere else in the table, the selection will be extended from the anchor cell to the cell that you clicked on. As was the case with the list widgets, the current item (identified by the currentRow and currentColumn attributes) is just the last cell that you clicked on.

It's important to recognize that the selection doesn't "grow" to include both the previously selected cells and the newly selected cells, unless they all happen to lie on the same side of the anchor. In other words, the selection in a table always pivots around the anchor cell. This may be a little counterintuitive at first (it was to me, anyways). The selStartRow, selEndRow, selStartColumn and selEndColumn attributes will always contain the starting and ending row and column indices for the entire selection, when there is one. Note that because of how the table's selection model works, one of those endpoints—either (selStartRow, selStartColumn) or (selEndRow, selEndColumn)—will be the anchor cell.

When the user clicks on a row heading, all of the cells in that row will become selected. Likewise, when the user clicks on a column heading, all of the cells in that column become selected. You can disable this behavior by setting either or both of the TABLE_NO_ROWSELECT and TABLE_NO_COLSELECT options.

```
# Disable row and column selections
table.tableStyle |= TABLE_NO_ROWSELECT|TABLE_NO_COLSELECT
```

The table sends a SEL_COMMAND message when you click on a table item, and the message data is an FXTablePos instance. An FXTablePos is just simple data object with row and col accessor methods, for reading the row and column of the selected table item.

You can, of course, programmatically set modify the selection, although you normally rely on the user to perform those actions interactively. Try

as you might, FXTable won't let you trick it into making a selection that violates its selection model. For example, the following code will result in only one cell—the one at (5, 5)—being selected.

```
table.selectItem(0, 0)
table.selectItem(5, 5)
```

To select a range of cells, use the selectRange() method.

```
# Select all of the cells between (0, 0) and (5, 5), inclusively
table.selectRange(0, 0, 5, 5)
```

As was the case for the FXTreeList, the easiest way to keep up with which table items are selected is to store them in an Array whose contents are updated in response to SEL_SELECTED and SEL_DESELECTED messages.

Download tableexample2.rb

```
selected_items = []
table.connect(SEL_SELECTED) do |sender, sel, pos|
  item = sender.getItem(pos.row, pos.col)
  selected_items << item unless selected_items.include? item
end
table.connect(SEL_DESELECTED) do |sender, sel, pos|
  selected_items.delete(sender.getItem(pos.row, pos.col))
end
```

This concludes our look at the widgets that FXRuby provides for dealing with collections of data, but there are a number of other, similar widgets in the library that you may want to take a look at as well. For example, the FXTable widget uses a pair of FXHeader widgets internally to display its row and column headings, but you can pull that widget out and use it by itself. The FXFoldingList is a sort of cross between an FXTreeList and an FXHeader that allows you to associate multiple columns of data with each item in a tree list. An FXIconList is used by the file dialog to provide several different kinds of views on a list of files, but you can also repurpose it to display other kinds of lists. You'll find documentation for each of these widgets in the FXRuby API documentation, and the standard FXRuby source distribution includes examples for each of them.

Next, we're going to shift gears and take a look at another one of FOX's more complicated widgets, the FXText widget, which you can use to edit large text documents.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

FXRuby's Home Page
http://pragprog.com/titles/fxruby
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/fxruby.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com