Prepared exclusively for RubyLearning's FXRuby participants

FXRuby: A Quick Start

Revised Edition – 4th Mar. 2009
First Edition – Mar. 2009

PLEASE SUPPORT RUBYLEARNING

Table of Contents

Acknowledgements 3
Introduction 4
Who is the eBook for? 4
Concept and Approach 4
How to Use This eBook 4
About the Conventions Used in This eBook 5
Resources 6
A Quick Look at FXRuby 7
Installing FXRuby 7
The Basics 8
A Little Optimization 10
Event Loop 10
Summary 11
Wondering what you can do with FXRuby? 13
Building a Simple Text Editor 14
Adding a Pull-down Menu 14
Adding a multi-line text component 17
Summary 18
Assignment 22
Some useful classes 22
FXTextField and FXDataTarget 22
FXButton 23
About the Author 26

Acknowledgements

There are a good number of people who deserve thanks for their help and the support they provided, either while or before this eBook was written, and there are still others whose help will come after the eBook is released.

I'd like to thank my 'gang' of assistant teachers, mentors and patrons at http://rubylearning.org/ for their help in making this eBook far better than I could have done alone.

I have made extensive references to the information related to FXRuby, available in the public domain (wikis, blogs and articles) – refer to the **Resources** chapter.

My acknowledgment and thanks to all of them.

I would like to thank Michael Kohl (of RubyLearning) for proofreading this eBook.

Introduction

Who is the eBook for?

This "**FXRuby: A Quick Start**" eBook is a starting point for people new to FXRuby and a guide to help learn it as quickly and easily as possible. By the time you finish this eBook, you will have a basic understanding of FXRuby and thus be ready to delve deeper into it. To work with FXRuby, some knowledge of the Ruby programming language is essential.

Concept and Approach

I have tried to organize this eBook so that each chapter builds upon the skills acquired in the previous chapters, to make sure that you will never be called upon to do something that you have not already learned. This eBook not only teaches you how to do something, but also provides you with the chance to put those morsels of knowledge into practice with exercises. It therefore contains several exercises, or assignments, to facilitate a hands-on learning approach.

How to Use This eBook

I recommend that you go through the entire eBook chapter by chapter, reading the text, running the provided examples and doing the assignments along the way. This will give you a much broader understanding of how things are done (and of how you can get things done), and it will reduce the chance of anxiety, confusion, and worse yet, mistakes.

It is best to read this eBook and complete its assignments when you are relaxed and have time to spare. Nothing makes things go wrong more than working in a rush. And keep in mind that the assignments in this eBook are fun, not just work exercises. So go ahead and enjoy them.

About the Conventions Used in This eBook

Explanatory notes (generally provide some hints or gives a more in-depth explanation of some point mentioned in the text) are shown shaded like this:

This is an explanatory note. You can skip it if you like - but if you do so, you may miss something of interest!

Any source code in this Ruby eBook, is written like this:

```
def hello
  puts "hello"
end
```

When there is a sample program to accompany the code, the program name is shown like this: **hello.rb**

Though we will be discussing FXRuby on the Windows platform, these notes should be appropriate for Linux/Mac OS X too unless explicitly stated otherwise.

Please remember that in Ruby, there's always more than one way to solve a given problem.

Finally, if you notice any errors or typos, or have any comments or suggestions or good exercises I could include, please email me at: **mail@satishtalim.com**.

Resources

The API documentation for FXRuby, is freely available – http://www.fxruby.org/doc/api/

Book: FXRuby: Create Lean and Mean GUIs with Ruby – http://www.pragprog.com/titles/fxruby/fxruby

A Quick Look at FXRuby

FXRuby is a library for developing graphical user interfaces (GUIs) for your Ruby applications. FXRruby is based on the popular open source C++ Library FOX, and exposes all its functionality. Fox and FXRuby are both licensed under LGPL which permits the use of those libraries in both free and proprietary (commercial) software applications – http://www.gnu.org/licenses/lgpl.html

This eBook introduces FXRuby to the course participants and is based on the various articles and tutorials on the FXRuby site – http://www.fxruby.org/

Installing FXRuby

For installing FXRuby, open a command window and type:
`c:\> sudo gem install fxruby`

For Windows, you need to type:
`c:\> gem install fxruby`

For installing Fox on *Mac OS X* –
http://www.fox-toolkit.net/documentation/building-fox-on-mac-os-x

If you have installed Ruby with MacPorts; use MacPorts to install FXRuby. The gem is made against OS X-built in Ruby.
`sudo port install rb-fxruby`

Supported platforms for Fox are:
http://fifthplanet.net/cgi-bin/wiki.pl?Supported_Platforms

For installing FXRuby on other operating systems , please refer to – http://www.fxruby.org/

The Basics

We will put together some basic code to display a "Welcome FXRuby Participants" window. All of the code associated with the FXRuby extension is provided by the fox16 gem, so we need to start by requiring it:

```
require 'fox16'
```

Since all of the FXRuby classes are defined under the Fox module, you'd normally need to refer to them using their "fully qualified" names (i.e. names that begin with the **Fox::** prefix). Because this can get a little tedious and name conflicts between FXRuby and other Ruby extensions are unlikely, we will include the Fox module in our global namespace:

```
require 'fox16'
include Fox
```

Every FXRuby program begins by creating an **FXApp** instance. **FXApp** is the glue that holds everything together and the most frequent way to start the application's main event loop (which you'll see later on in this eBook):

```
require 'fox16'
include Fox
app = FXApp.new
```

The next step is to create an **FXMainWindow** instance to serve as the application's main window. We pass our newly created **app** object as the first argument to **FXMainWindow.new** to associate the new **FXMainWindow** instance with this **FXApp**, followed by the window title, width and height:

```
require 'fox16'
include Fox
app = FXApp.new
main = FXMainWindow.new(app, "Welcome FXRuby Participants", :width
=> 350, :height => 100)
```

So far we only created the client-side objects, but FOX distinguishes between client-side and server-side data. To create the server-side objects associated with the already-constructed client-side objects, we call **FXApp#create**:

```
require 'fox16'
include Fox
app = FXApp.new
main = FXMainWindow.new(app, "Welcome FXRuby Participants", :width
=> 350, :height => 100)
app.create
```

By default, all windows in FXRuby programs are invisible, so we need to call our main window's show instance method. **PLACEMENT_SCREEN** is a request to centre the window on the screen:

```
require 'fox16'
include Fox
app = FXApp.new
main = FXMainWindow.new(app, "Welcome FXRuby Participants", :width
=> 350, :height => 100)
app.create
main.show(PLACEMENT_SCREEN)
```

The last step is to start the program's main loop by calling the instance method '**run**' of our application object. The **FXApp#run** method doesn't return until the program exits:

```
require 'fox16'
include Fox
app = FXApp.new
main = FXMainWindow.new(app, "Welcome FXRuby Participants", :width
=> 350, :height => 100)
app.create
main.show(PLACEMENT_SCREEN)
app.run
```

At this point, we have a working (if not very interesting) program that uses FXRuby. Let's store this program in a file – **fx1.rb**. The screenshot shows you how the window looks –

A Little Optimization

Let's make the application's main window a subclass of **FXMainWindow**:

```
require 'fox16'
include Fox
class WelcomeWindow < FXMainWindow
  def initialize(app, title, w, h)
    super(app, title, :width => w, :height => h)
  end
  def create
    super
    show(PLACEMENT_SCREEN)
  end
end
app = FXApp.new
WelcomeWindow.new(app, "Welcome FXRuby Participants", 350, 100)
app.create
app.run
```

It becomes convenient to focus the application control inside a custom main window class, as shown above.

Event Loop

FXRuby programs are event-driven. After initialization, FXRuby programs enter the event loop: they wait for an event to occur, respond to it and then resume waiting for the next event. FXRuby models an event as one object, the *sender*, sending a message to another object, the *target*. Every message that's sent from one FXRuby object to another consists of a *message type*, a *message identifier*, and some *message data*. The message type is a constant

whose name begins with **SEL_**. The message identifier is also a constant, and it's used by the target (the receiver of the message) to distinguish between different incoming messages of the same type. The message data is just an object that provides some additional context for the message. The most useful message that a widget sends to its target is its **SEL_COMMAND** message. The specific meaning of **SEL_COMMAND** is different for different widgets. The **connect()** method connects messages sent from a widget to a chunk of code that handles them. Under the hood, the **connect()** method creates a target object and message identifier and then assigns those to the message sender:

```
exit_cmd.connect(SEL_COMMAND) do |sender, selector, data|
  # Handle it here
end
```

In the above example, we "connect" the **SEL_COMMAND** message from the **exit_cmd** to a Ruby block that expects three arguments. You can use any names for these arguments. In the above example 'sender' is a reference to the object which sent the message (**exit_cmd**), 'selector' is a value that combines the message type and identifier and 'data' is a reference to the actual message data.

Summary

(Thanks to Michèle Garoche for these notes.)

1. For non-Windows platforms, the most direct way to make RubyGems available is to just require it in the source code:

```
require 'rubygems'
```

Please note that it would be better to "**Set Up the RubyGems Environment**" as shown here –

http://docs.rubygems.org/read/chapter/3#page70

2. All of the code associated with the FXRuby extension is provided by the fox16 gem, so we need to start by requiring this gem:

```
require 'fox16'
```

3. We include the Fox module into our global namespace like this:

```
include Fox
```

4. There are many classes in FXRuby but you don't need to know them all in depth. The *trick* here is to read the api in the order they are going to be used in your program.

5. For most FXRuby programs you need to create a **FXMainWindow** instance to serve as an application's main window. Remember to associate **FXMainWindow** with **FXApp**.

6. To create the server-side objects associated with the already-constructed client-side objects, we call the **FXApp.create** method.

7. It is essential to understand that everything inside a graphical user interface is event driven, so make sure you fully understand events and the event loop.

8. Read and reread the explanations (not the code) in the tutorial until you understand them.

9. Nearly all graphical elements (buttons, text fields, menu bar, menu elements, etc.) have a state (active, non active), that indicates to the user whether the graphical element is accessible to the user or not.

10. The access to those elements is driven by events. An event is anything that sends a given message to the application or an element of the application. Events include clicks on buttons or menu elements, key strokes in text fields, mouse movement, drag and drop actions, etc., all of which may also be programmatically generated. More on **Event Driven Programming** –

http://en.wikipedia.org/wiki/Event-driven_programming

11. When an FXRuby program enters the main event loop, it waits for an event to occur, responds accordingly and then resumes waiting for events.

12. Every message that's sent from one FXRuby object to another consists of a message type, a message identifier, and some message data.

13. The message type is a constant whose name begins with **SEL_**. The message identifier is also a constant, and it's used by the target (the receiver of the message) to distinguish between different incoming messages of the same type. The message data is just an object that provides some additional context for the message.

14. The *most useful message* that a widget sends to its target is its **SEL_COMMAND** message. The specific meaning of **SEL_COMMAND** changes according to the widget.

15. The **`connect()`** method connects messages sent from a widget to a chunk of code that handles them.

Wondering what you can do with FXRuby?

To see what can be done with FXRuby you should have a look at the following URL:

http://www.fxruby.org/doc/examples.html

Building a Simple Text Editor

We'll now build a simple text editor. For defining the basic structure of our application we will define a TextEditor class as a subclass of **FXMainWindow**:

```
# texteditor.rb
require 'fox16'
include Fox
class TextEditor < FXMainWindow
  def initialize(app, title, w, h)
    super(app, title, :width => w, :height => h)
  end
  def create
    super
    show(PLACEMENT_SCREEN)
  end
end
app = FXApp.new
TextEditor.new(app, "Simple Text Editor", 600, 400)
app.create
app.run
```

Adding a Pull-down Menu

We will use the **FXMenuBar, FXMenuPane, FXMenuTitle** and **FXMenuCommand** classes together to create a menu system (with pull-down menus) for our application.

Let us put all the code related to constructing the menu bar in a new private instance method named **add_menu_bar()** –

```
def add_menu_bar
  menu_bar = FXMenuBar.new(self, LAYOUT_SIDE_TOP |
LAYOUT_FILL_X)
end
```

We are going to use a "nonfloatable" menu bar, and it has two arguments: the first is the parent window i.e. self which refers to the TextEditor window, since this is an instance method of the TextEditor

class; the second argument is an options value that tells FXRuby to place the menu bar at the top of the main window's content area and to stretch it as wide as possible.

Next we create an **FXMenuPane** window, as a child of **FXMenuBar**:

```
file_menu = FXMenuPane.new(self)
```

A menu pane is a kind of pop-up window. You interact with it by choosing a menu command, and then it "pops down" again. You call a menu pane by clicking the **FXMenuTitle** widget associated with that menu pane.

```
FXMenuTitle.new(menu_bar, "File", :popupMenu => file_menu)
```

The **FXMenuTitle** is a child of **FXMenuBar**, but it also needs to know which menu pane it should display when it is activated, and so we pass that in as the :popupMenu argument. Next we add our command:

```
load_cmd = FXMenuCommand.new(file_menu, "Load")
load_cmd.connect(SEL_COMMAND) do
  # ...
end
```

By calling **connect()** on **load_cmd**, we are associating a block of Ruby code with that command. When the user selects the Load command from the File menu, we want to display a file selection dialog box. Here is what should go inside the **connect()** block:

```
dialog = FXFileDialog.new(self, "Load a File")
dialog.selectMode = SELECTFILE_EXISTING
dialog.patternList = ["All Files (*)"]
if dialog.execute != 0
  load_file(dialog.filename)
end
```

We start by constructing an **FXFileDialog** as a child of the main window, with the title "Load a File". Next, we set the file selection mode to **SELECTFILE_EXISTING**, which means the user can select

only an existing file. We also initialize the **patternList** to an array of strings that indicate the available file filters. Finally, we call **execute()** to display the dialog box and wait for the user to select a file. The **execute()** method for a dialog box returns a completion code of either 0 or 1, depending upon whether the user clicked Cancel to dismiss the dialog box or OK to accept the selected file. If the user clicked Cancel, we don't need to do anything else for this command. Otherwise, we want to call the as-yet nonexistent **load_file()** private method to load the selected file.

Similarly, we can write code for **new_cmd, save_cmd** and **exit_cmd**. We will add the **FXMenuSeparator** widget to a menu pane to create a visual break between groups of related command.

```
FXMenuSeparator.new(file_menu)
```

Finally, we add a call to the **add_menu_bar()** method from the **initialize()** method. The code so far:

```
# texteditor.rb
require 'fox16'
include Fox
class TextEditor < FXMainWindow
  def initialize(app, title, w, h)
    super(app, title, :width => w, :height => h)
    add_menu_bar
  end

  def create
    super
    show(PLACEMENT_SCREEN)
  end

  private
  def add_menu_bar
    menu_bar = FXMenuBar.new(self, LAYOUT_SIDE_TOP |
LAYOUT_FILL_X)
    file_menu = FXMenuPane.new(self)
    FXMenuTitle.new(menu_bar, "File", :popupMenu => file_menu)
    new_cmd = FXMenuCommand.new(file_menu, "New")
    new_cmd.connect(SEL_COMMAND) do
      #
    end
```

```
    load_cmd = FXMenuCommand.new(file_menu, "Load")
    load_cmd.connect(SEL_COMMAND) do
      dialog = FXFileDialog.new(self, "Load a File")
      dialog.selectMode = SELECTFILE_EXISTING
      dialog.patternList = ["All Files (*)"]
      if dialog.execute != 0
        load_file(dialog.filename)
      end
    end
    save_cmd = FXMenuCommand.new(file_menu, "Save")
    save_cmd.connect(SEL_COMMAND) do
      dialog = FXFileDialog.new(self, "Save a File")
      dialog.selectMode = SELECTFILE_ANY
      dialog.patternList = ["All Files (*)"]
      if dialog.execute != 0
        save_file(dialog.filename)
      end
    end
    FXMenuSeparator.new(file_menu)
    exit_cmd = FXMenuCommand.new(file_menu, "Exit")
    exit_cmd.connect(SEL_COMMAND) do
      exit
    end
  end

  def load_file(filename)
    contents = ""
    File.open(filename, 'r') do |f1|
      while line = f1.gets
        contents += line
      end
    end
    puts contents
  end

  def save_file(filename)
    File.open(filename, 'w') { |file|   file.write(@txt.text) }
  end
end

app = FXApp.new
TextEditor.new(app, "Simple Text Editor", 600, 400)
app.create
app.run
```

Adding a multi-line text component

FXText is a fully featured text-editing component to add to our application.

```
def add_text_area
  @txt = FXText.new(self, :opts =>
TEXT_WORDWRAP|LAYOUT_FILL)
  @txt.text = ""
end
```

As shown above, we shall add a private method **add_text_area** to our class. By default, the text buffer for an **FXText** widget is empty. You can initialize its value by assigning a string to its text attribute. Finally, we add a call to the **add_text_area()** method from the **initialize()** method.

Now let's add some functionality to new_cmd.

```
new_cmd.connect(SEL_COMMAND) do
  @txt.text = ""
end
```

Next, let's modify the functionality of the **load_file()** method:

```
def load_file(filename)
  contents = ""
  File.open(filename, 'r') do |f1|
    while line = f1.gets
      contents += line
    end
  end
  @txt.text = contents
end
```

Summary

Here are the classes and their hierarchy as used in the Text Editor:

Object
FXApp (it can send messages to objects for a few special events)
(http://www.fxruby.org/doc/api/classes/Fox/FXApp.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXComposite (Base composite)
FXShell (The Shell widget is used as the base class for top level windows)
FXTopWindow (Abstract base class for all top-level windows)
FXMainWindow (The Main Window is usually the central window of an application)
(http://www.fxruby.org/doc/api/classes/Fox/FXMainWindow.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXComposite
FXPacker
FXDockBar
FXToolBar
FXMenuBar (Menu bar)
(http://www.fxruby.org/doc/api/classes/Fox/FXMenuBar.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXComposite
FXShell
FXPopup
FXMenuPane (Popup menu pane)
(http://www.fxruby.org/doc/api/classes/Fox/FXMenuPane.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXMenuCaption
FXMenuTitle
(http://www.fxruby.org/doc/api/classes/Fox/FXMenuTitle.html)

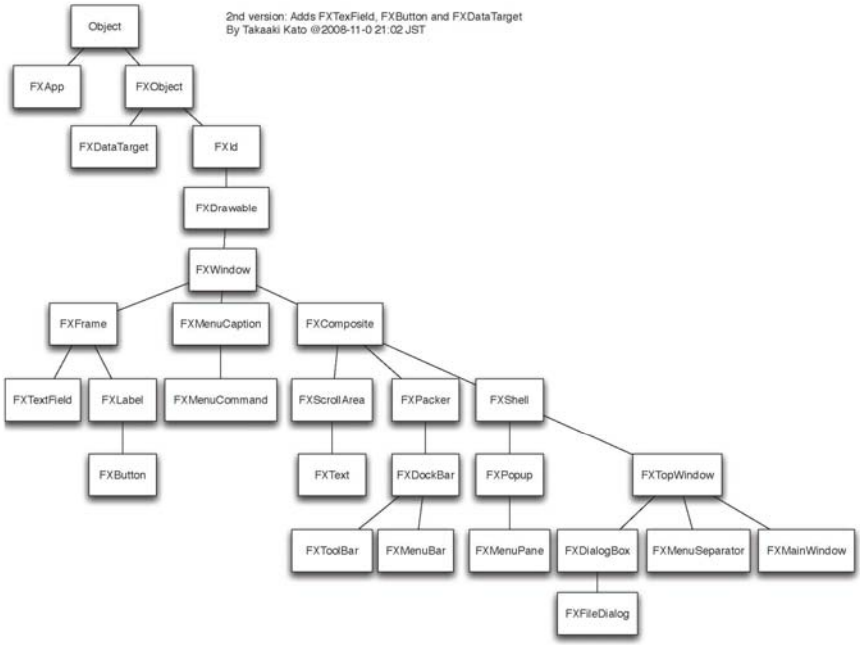
Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXMenuCaption
FXMenuCommand
(http://www.fxruby.org/doc/api/classes/Fox/FXMenuCommand.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXMenuSeparator (used to delineate items in a popup menu)
(http://www.fxruby.org/doc/api/classes/Fox/FXMenuSeparator.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXComposite (Base composite)
FXShell (The Shell widget is used as the base class for top level windows)
FXTopWindow (Abstract base class for all top-level windows)
FXDialogBox
FXFileDialog
(http://www.fxruby.org/doc/api/classes/Fox/FXFileDialog.html)

Object
FXObject (the base class for all objects in Fox)
FXId (Encapsulates a server side resource)
FXDrawable (abstract base class for any surface that can be drawn upon)
FXWindow (base class of all Fox GUI widgets such as buttons and sliders)
FXComposite (Base composite)
FXScrollArea
FXText (supports editing of multiple lines of text.)
(http://www.fxruby.org/doc/api/classes/Fox/FXText.html)

The same is shown graphically -



Assignment

Using FXRuby, write a program and call it **myproject.rb**. The screen should look like this:

Write the class **MyProject < FXMainWindow** for this assignment.

Some useful classes

You could be using some of these classes in the above assignment.

FXTextField and FXDataTarget

This is used to provide for the input and subsequent editing of single-line text strings. The **FXTextField** sends a **SEL_COMMAND** message to its target when the user presses the Return or Enter key after typing some text in a text field. It will also send a **SEL_COMMAND** message when the text field loses the keyboard focus (because the user clicked somewhere else or pressed the Tab key to shift the focus to some other widget). You can associate a text field with a data target – **FXDataTarget**, as follows:

```
@name_target = FXDataTarget.new("Satish")
name_text = FXTextField.new(p, 25,
      :target => @name_target, :selector =>
FXDataTarget::ID_VALUE, :opts => FXTextField::TEXTFIELD_NORMAL)
@name_target.connect(SEL_COMMAND) do
  puts "The name is #{@name_target.value}"
end
```

The **new** method of **FXTextField** takes five parameters:

```
FXTextField.new(p, ncols, target, selector, opts)
```

Where:
p - the parent window for this text field
ncols - number of visible items [Integer]
target - the message target, if any, for this text field
selector - the message identifier for this text field [Integer]
opts - text field options [Integer] like
FXTextField::TEXTFIELD_NORMAL,
FXTextField::TEXTFIELD_READONLY,
FXTextField::TEXTFIELD_PASSWD.

The **FXDataTarget** is a special kind of object that allows a valuator widget like an **FXTextField** to be directly connected with a variable in the program and knows how to do the right thing in response to certain message types such as **SEL_COMMAND**. Whenever the valuator widget controls changes, the variable connected through the data target is automatically updated; conversely, whenever the program changes a variable, all the connected valuator widgets will be updated to reflect this new value on the display. **FXDataTarget::ID_VALUE** causes the **FXDataTarget** to ask sender for values.

FXButton

This class provides a push button, with an optional icon and/or text label. When pressed, the button widget sends a **SEL_COMMAND** to its target.

The **new** method of **FXButton** takes three parameters:

```
FXButton.new(p, text, opts)
```

Where:
p – parent

text – text on the button
opts – default is **BUTTON_NORMAL**

Let's create a button to quit our application by adding a message handler for FXButton:

```
theButton.connect(SEL_COMMAND) do |sender, selector, data|
  exit
end
```

Fox messages have four important elements:

The message ***sender*** is the object that sends the message. In this case, the **FXButton** instance is the sender.

The message *type* is a predefined integer constant that indicates what kind of event has occurred (i.e. why this message is being sent). In this case, the message type is **SEL_COMMAND**, which indicates that the command associated with this widget should be invoked.

The message *identifier* is another integer constant that is used to distinguish between different messages of the same type. For example, the message that tells a Fox window to make itself visible is a **SEL_COMMAND** message with the identifier **FXWindow::ID_SHOW** (where **ID_SHOW** is a constant defined in the **FXWindow** class). A different message identifier, **FXWindow::ID_HIDE**, tells an **FXWindow** instance to make itself invisible.

The message *data* is an object containing message-specific information. For this case (the FXButton's **SEL_COMMAND** message, there is no interesting message data).

For historical reasons, the message type and identifier are usually packed into a single 32-bit unsigned integer known as the ***selector***, and this is the value that is passed into the message handler block. Since we don't actually need to use the ***sender, selector*** or ***data*** arguments for this particular message handler, we can just ignore them and shorten the code to:

```
theButton.connect(SEL_COMMAND) { exit }
```

--- The End ---

About the Author

Satish Talim (http://satishtalim.com/) is a senior software consultant based in Pune, India with over 30+ years of I.T. experience. His experience lies in developing and executing business for high technology and manufacturing industry customers. Personally his strengths lie in Business Development and Business Networking apart from new product and solution ideas. Good experience of organization development. Excellent cross disciplinary background in engineering, computer science and management.

He -

- Has helped start subsidiaries for many US based software companies like **Infonox** based in San Jose, CA
http://www.infonox.com/default.shtml
Maybole Technologies Pvt. Ltd.
http://servient.com/
subsidiary of Servient Inc. (based in Houston, Texas) in Pune, India.

- Has been associated with Java / J2EE since 1995 and involved with Ruby and Ruby on Rails since 2005.
also started and manages two very active Java and Ruby User Groups in Pune, India - **PuneJava**
http://tech.groups.hotmail.com/group/pune-java/
and **PuneRuby**
http://tech.groups.hotmail.com/group/puneruby/

- Is a **Ruby Mentor**
http://rubymentor.rubyforge.org/wiki/wiki.pl?AvailablePureRubyMentors
on rubyforge.org, helping people with Ruby programming.

He lives in Pune, India, with his wife, son and his Labrador Benzy. In his limited spare time he enjoys traveling, photography and playing online chess.