Ruby and Python Programming Topics for Mac OS X

Tools & Languages: Other Languages

2007-10-31

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Ruby and Python Programming Topics for Mac OS X

Ruby and Python are two popular scripting languages that, with Mac OS X version 10.5, are becoming even more significant alternatives for software development on Mac OS X, especially with additional support for bridges between the scripting languages and Cocoa and Open Scripting Architecture (OSA),

This collection of articles describes the Ruby and Python resources of Mac OS X, and shows how to use some of those programming resources. This document is intended for Python and Ruby developers who are new to Mac OS X as well as for experienced Cocoa and Carbon developers who are relatively new to the scripting languages.

Organization of This Document

These programming topics consist of the following articles:

- "Ruby and Python on Mac OS X" (page 9) gives an overview of the Ruby and Python resources on Mac OS X, including the Cocoa and OSA bridges, and provides links to websites related to these scripting languages.
- "Building a RubyCocoa Application: A Tutorial" (page 19) steps you through the creation of a simple RubyCocoa application.
- "Using Scripting Bridge in PyObjC and RubyCocoa Code" (page 43) describes how to incorporate Scripting Bridge in your code to communicate with and control OSA-compliant applications.
- "Generating Framework Metadata" (page 51) explains what framework metadata is and how to create these XML files used by the scripting bridges.

See Also

The article "Using PyObjC for Developing Cocoa Applications With Python" provides an excellent introduction to using PyObjC.

Ruby and Python on Mac OS X

Ruby and Python, two immensely popular object-oriented scripting languages, have been installed as part of Mac OS X for many years now. But their relevance to software development, and especially application development, assumes even greater importance in Mac OS X v10.5. The following sections summarize the capabilities and components of Ruby and Python and describe the bridges being developed and enhanced for Mac OS X to support Cocoa programming and AppleScript-command processing from those scripting languages.

What Are Ruby and Python?

Ruby and Python are interpreted object-oriented scripting languages. As interpreted languages, you can change and run code immediately, without having to wait for the code to compile. Python and Ruby also have all the features one would expect to find in dynamic object-oriented programming languages, such as inheritance, encapsulation, introspection, and subclassing. The syntax of both languages is simple, compact, and consistent, and supports both regular expressions and sophisticated string manipulations. Memory management is built into both languages; garbage collectors automatically free memory occupied by unneeded objects. With both Python and Ruby you can call operating system routines directly. They offer ways to extend their native capabilities, including C-language interfaces.

Although their similarities are striking, these scripting languages do have some differences. While Python code can contain both objects and built-in types, in Ruby everything is an object. There are no primitive or built-in types, such as integers. Thus anything in Ruby code can accept messages. And you don't have to declare variables to be of specific object types. To distinguish variables as global, local, instance, and class, Ruby uses naming conventions. Ruby also has mix-in by modules and blocks, language features absent in Python.

Beyond the similarities of languages and interpreters, Python and Ruby share other things in common. Both have extensive standard libraries of classes and modules. Both scripting languages can be used in a wide variety of software projects, including system programming (command-line utilities and daemons), user-interface design, Internet and networking tasks, database programming, component integration, and, of course, rapid prototyping. And both are the products of open-source projects supported by large and enthusiastic developer communities.

Both languages come with a basic set of command-line utilities. In addition to the interactive interpreter, `irb`, Ruby includes `ri` and `rdoc` (for displaying and generating documentation, respectively), `erb` (for interpreting files with embedded Ruby code), and `testrb` (for running test suites on Ruby code). In addition to the language interpreter, `python`, Python includes `pydoc` for viewing documentation and `pythonw` for running Python scripts that display a graphical user interface. All of these utilities are located in `/usr/bin`.

The Standard Ruby Package

On Mac OS X Ruby includes more than the language interpreter and documentation and testing utilities. A standard installation offers the following Ruby-related services, frameworks, and protocols:

- RubyGems—A package manager for Ruby
- rake—A `make`-like utility for Ruby scripts
- Rails (or Ruby on Rails)—A framework for creating database-backed web applications with designs conforming to the Model-View-Controller pattern

  For more information on Ruby on Rails, go to http://developer.apple.com/tools/rubyonrails.html.
- Mongrel—A fast HTTP library and server used for hosting Ruby web applications
- Capistrano—A framework and utility for executing commands in parallel on multiple remote machines, via SSH, primarily to expedite the deployment of web applications
- Ferret—A search engine
- OpenID—A service that provides OpenID identification to Ruby programs
- sqlite3-ruby—A module that enables Ruby scripts to interact with a SQLite3 database
- libxml-ruby—A module for reading and writing XML documents using Ruby
- dnssd—Ruby interface for DNS Service Discovery (that is, Bonjour)
- net-ssh and net-sftp—Pure Ruby implementations of the SSH and SFTP client protocols

The Standard Python Package

The Python modules included in the standard package for Mac OS X are the following:

- altgraph — Python graph (network) package
- bdist_mpkg — Builds Mac OS X installer packages from distutils
- macholib — Mach-O header analysis and editing
- modulegraph — Python module dependency analysis tool
- numpy (or NumPy) — Array processing for numbers, strings, records, and objects
- py2app — Creates standalone Mac OS X applications with Python
- setuptools — Downloads, builds, installs, upgrades, and uninstalls Python packages
- xattr — A Python wrapper for Darwin's extended filesystem attributes

Except for `numpy` and `xattr`, all of these modules are used by PyObjC.

On-line Resources

You can find out more about Python from the following websites:

- Main Python website: http://www.python.org/
- Documentation: http://docs.python.org/
- Other developer resources: http://www.python.org/dev/

On-line resources for Ruby include the following websites:

- Documentation, downloads, and other resources: http://www.ruby-lang.org/
- Libraries: http://rubyforge.org/
- *why's (poignant) guide to Ruby* (http://poignantguide.net/ruby/), a whimsical, cartoon-illustrated introduction to Ruby

Bridges for Cocoa Development

Both Ruby and Python include bridges to the Objective-C runtime. Although these bridges are open-source projects, some changes have been made the implementation and tool support on Mac OS X v10.5 and later systems.

RubyCocoa

Because Ruby and Objective C share a common ancestor in Smalltalk, creating a bridge between them was relatively straightforward. RubyCocoa is a bridge that makes it possible for Ruby scripts to access Objective-C objects defined in frameworks and local project code. Consequently, one can do Cocoa programming in a Ruby script. RubyCocoa works by creating—automatically and upon demand—Ruby proxy objects that are bridged to Objective-C classes. It also forwards Ruby messages to the instances of these Objective-C classes. You can have a Cocoa application project that mixes Ruby and Objective-C source files. RubyCocoa supports all important features of Cocoa, such as key-value coding, key-value observing, Core Data, the document architecture, notifications, and undo management.

Note: For more information about RubyCocoa, go to rubycocoa.sourceforge.net. You can find RubyCocoa documentation and coding examples at http://rubycocoa.sourceforge.net/.

The following line of code creates a Ruby proxy class that wraps the Cocoa class NSButton:

```
OSX::NSButton
```

A message sent to an instance of this class is forwarded to the Objective-C instance within the proxy object. (If the object doesn't respond to the message, then RubyCocoa raises a runtime error.) As illustration, consider the following lines of Objective-C code:

```
// the NSRect structure (rect) is specified earlier
NSButton *button = [[NSButton alloc] initWithFrame:rect];
[button setTarget:self];
[button setAction:@selector(doGoodThings:)];
[button setEnabled:YES];
[view addSubview:button];
[button release];
```

In RubyCocoa, the equivalent to these lines would be the following:

```
button = NSButton.alloc.initWithFrame_(rect)
button.setTarget_(self)
button.setAction_(:doGoodThings)
button.setEnabled_(true)
view.addSubview_(button)
```

As you can see, RubyCocoa uses keypath-style dot notation is used to indicate (potentially nested) message invocations, starting with the object or class initiating the invocations. Note that the `release` is omitted in the RubyCocoa code snippet because the garbage collector takes care of object disposal.

The snippet of RubyCocoa code above uses the default messaging syntax, where underscores replace the colons of the Objective-C keywords . But RubyCocoa supports a variant of the default syntax that omits the final underscore. Thus, the two message syntaxes are:

```
# Default calling syntax
NSURL.alloc.initWithScheme_host_path_('http', 'localhost', 'sample')
# Same, but no underscore for final keyword
NSURL.alloc.initWithScheme_host_path('http', 'localhost', 'sample')
```

In a standard Mac OS X installation, the second syntax is disabled. However, you can enable it by setting the `OSX.relaxed_syntax` flag to `true`.

RubyCocoa takes care of object type conversions for you. When you pass parameters to a Ruby proxy object, RubyCocoa automatically converts the more basic Ruby types to proxies representing their Objective-C counterparts (for example, Ruby strings and `NSString` objects). It also converts objects returned from the Objective-C side to Ruby objects that act as proxies to those Objective-C objects. On the Ruby side, these proxy objects have more or less the same interfaces as their Ruby equivalents.

RubyCocoa adds several Xcode templates for building RubyCocoa applications of various types. The templates make it unnecessary for developers to create applications by writing RubyCocoa code using a shell editor (for example, Emacs or `vi`) and then manually constructing the various pieces of the application bundle. The Xcode templates make sure the application project is properly set up for RubyCocoa and that the application executable and its bundle are properly built. And they let you access the conveniences of a first-class integrated development environment. You can also design your user interfaces using the Interface Builder application. Currently there are four RubyCocoa application templates:

- Cocoa-Ruby applications (single window)
- Cocoa-Ruby document-based applications
- Cocoa-Ruby Core Data applications
- Cocoa-Ruby Core Data document-based applications

In addition to the project templates, RubyCocoa adds support for test units. In Xcode you can create a test-unit file by choosing New File from the File menu and then selecting “Ruby test case class” under the Ruby category in the New File Assistant. You can also set up a test-unit target by choosing “New Target” from the Project menu and then selecting “Unit Test Target” option in the New Target Assistant.

Apple’s implementation of RubyCocoa adds some features and makes some performance improvements, including the following:

- Apple has added support for generating metadata about the C-language parts of a framework’s Objective-C API.

  RubyCocoa can extract most of the information it needs about object-oriented symbols (such as classes and methods) from frameworks at runtime. Unfortunately, there is no purely dynamic way to introspect framework data that is C-based, such as constants, enumerations, and functions. To resolve this problem (in a way that avoids generating static code at build time), RubyCocoa reads a per-framework metadata file, which it loads at runtime. A command-line tool generates most of this metadata XML automatically but the framework developer may have to specify certain items manually, such as pass-by-reference parameters.. See “Generating Framework Metadata” (page 51) for more information on framework metadata and instructions on how to create the metadata description.

- Apple has made many performance improvements, involving the following:
  - RubyCocoa uses the `libffi` library for function calling and message dispatch.

    Instead of a message-dispatch implementation based on `objc_msgSend` or `NSInvocation`, RubyCocoa uses the `libffi` library from the GCC project. `libffi` makes it possible to call an arbitrary C function in a processor-agnostic way. It provides more scalability and better performance than the other alternatives. RubyCocoa also uses `libffi` when overriding or registering an Objective-C method implemented in Ruby, and when converting Ruby closures to C function pointers.
  - RubyCocoa efficiently copies objects as they cross the bridge either way.
  - RubyCocoa efficiently looks up selectors and classes.
- Apple's RubyCocoa accurately translates the Objective-C class hierarchy when it creates Ruby proxy objects, taking into account those classes that can be toll-free bridged to the Core Foundation counterparts

PyObjC

PyObjC is a bridge that lets you write sophisticated Cocoa applications using the Python scripting language. It enables Python objects to send messages to Objective-C objects and vice versa. With PyObjC you're not limited to the core Cocoa frameworks, Foundation and Application Kit. You can use any Objective-C framework from Python, and your projects can be a mix of Objective-C, C, and C++ code. PyObjC also supports full introspection of Objective-C classes and direct invocation of Objective-C APIs from the interactive interpreter. Like RubyCocoa, PyObjC incorporates supports the full range of Cocoa features such as key-value coding, key-value observing, Core Data, document-based applications, notifications, and undo management..

Note: The official PyObjC website is http://pyobjc.sourceforge.net/ and you can also get information on the current version is at http://www.python.org/pypi/pyobjc/1.3.5. You can find documentation, coding examples, downloadable installer packages, and other resources at (http://pyobjc.sourceforge.net/documentation/index.html) . A description of the standard distribution of PyObjC on Mac OSX is on the ADC website at http://developer.apple.com/cocoa/pyobjc.html.

PyObjC is useful for more than just Cocoa application (GUI) development. You can also use PyObjC for rapid prototyping of projects, and for writing Foundation-based command-line tools, screen savers, preference panes, and other forms of software.

PyObjC leaves little that is unbridged between Objective-C and Python. Objective-C classes can inherit from Python classes, and Python classes can inherit from Objective-C classes. You can declare categories on Objective-C classes, with the method definition given in Python. Python classes can implement and define Objective-C protocols, and it's possible to establish a binding between a Python object and an Objective-C object in Interface Builder.

In PyObjC, Cocoa classes are treated as normal Python classes, but (for Python programmers) with a somewhat different naming scheme for methods . The PyObjC equivalent of the RubyCocoa button code above is:

```
button = NSButton.alloc().initWithFrame_(rect)
button.setTarget_(self)
button.setAction_('doGoodThings:')
button.setEnabled_(True)
view.addSubView_(button)
```

PyObjC performs a simple translation from Objective-C selector names to Python method names (and vice versa when new methods are defined), replacing all colons by underscores. This is the only messaging syntax supported.

PyObjC automatically converts Python objects passed to the Objective-C runtime to the correct Objective-C type, and also converts Objective-C objects passed back into Python. For example, Python strings are proxied using an NSString subclass when they are passed to Objective-C code; likewise, an NSString object is proxied using a Python unicode-subclass when the object passes into Python. Unlike RubyCocoa, predicates work without further work on your part; in other words, `if button.isEnabled: doSomething()`) works as one would expect.

PyObjC's support for pass-by-reference arguments is similar to that for RubyCocoa, and predates it by many years. You can learn more about the exact semantics in the introductory documentation for PyObjC (http://pyobjc.sourceforge.net/documentation/pyobjc-core/intro.html).

A change in the Leopard version of PyObjC is that it uses the same XML metadata description as does RubyCocoa (see "RubyCocoa" (page 11) for an overview). Another change is that PyObjC now supports all Core Foundation–based types as well, not only those that can be toll-free bridged to Cocoa classes.

The open-source version of PyObjC includes a number of Xcode templates that make it easy to create and configure Cocoa-Python application projects. By using the templates, you can have the development environment for your project set up for you; it eliminates the need to code using a shell editor or text processor and then manually construct the various parts of the application bundle. You can compose your user interfaces using Interface Builder and then save them to a nib file. And you have access to a sophisticated integrated development environment with features such as multiple build targets and symbol and documentation look-up. Four PyObjC application templates are offered:

- Cocoa-Python applications (single window)
- Cocoa-Python document-based applications
- Cocoa-Python Core Data applications
- Cocoa-Python Core Data document-based applications

The Apple version of PyObjC for Mac OS X version 10.5 includes two additional improvements:

- PyObjC uses the same the same XML metadata scheme as RubyCocoa to define the non-object-oriented parts of a framework..
- PyObjC supports all Core Foundation opaque types and not only those that can be toll-free bridged to Cocoa classes..

The Advantages of PyObjC and RubyCocoa

The RubyCocoa and Python bridges bring several advantages to Cocoa development, both for experienced Ruby and Python "scripters" and for Objective-C developers. By letting you mix and match Objective-C, Ruby, and Python, the bridges give you the option of choosing the best language tool for whatever programming goal you have. At the same time, they give your code access to Cocoa technologies such as bindings and Core Data. Moreover, your RubyCocoa and PyObjC projects can use the capable project management of Xcode and the rapid interface development offered by Interface Builder.

By bridging the Ruby and Python languages to the Objective-C runtime, PyObjC and RubyCocoa open the door to Cocoa application development for thousands of Python and Ruby scripters. But they also offer benefits to experienced Objective-C developers. If you are such a developer, you can take advantage of both scripting languages' sophisticated regular-expression features for textual processing. You also have access to the extensive libraries for both Python and Ruby. The interpretive nature of RubyCocoa and PyObjC means you can use them for rapid application prototyping to help you locate design problems early in the development cycle. Using the interpreter, you can inject code into your application on the fly and instantly inspect and manipulate objects in your application.

The bridges' conjunction of two object-oriented languages—Ruby and Python on one side and Objective-C on the other—enables even more dynamism than any of the languages provides on its own. For example, with PyObjC you can create Cocoa-compatible classes at runtime and even create new methods while your application continues to execute.

A final advantage of RubyCocoa and PyObjC is that they are extensions of languages that run on a variety of systems, including Linux and Windows. In other words, they are cross-platform. You could thus maintain a cross-platform code base in Ruby or Python—your model objects, as it were—and use the bridged version of the language to control the user interface and manage the application.

Bridges for OSA Automation

You have several options for writing Ruby or Python scripts that can communicate with scriptable applications, enabling them to control those applications and exchange data with them. These technologies are bridges to the Open Scripting Architecture (OSA) infrastructure, which uses Apple events for interprocess communication. The native solution is Scripting Bridge, which is a bridge to the Objective-C runtime and thus can be used in RubyCocoa or PyObjC scripts. You also use open-source Ruby and Python bridges to OSA, and thereby merge the power of Ruby or Python with that of AppleScript and Apple event processing.

Scripting Bridge

Many applications installed on Mac OS X are scriptable. Through the Scripting Bridge technology, RubyCocoa and PyObjC scripts and programs can communicate with these applications, controlling them and exchanging data with them. For example, using Scripting Bridge a RubyCocoa script could select and play music tracks in iTunes; or it could search a mailbox (maintained by the Mail application) for messages with a certain phrase and put those messages into a new TextEdit document.

Scriptable applications define a interface through which they can respond to Apple events, which are a part of the Open Scripting Architecture (OSA). Apple events frequently originate in AppleScript scripts and make use of the Apple Event Manager of OSA as the mechanism of delivery. Scripting Bridge is a framework that implements an Objective-C bridge to OSA-compliant applications—that is to say, applications having a scripting interface that follows the guidelines described in Technical Note T2106 and *Cocoa Scripting Guide*. It enables programs written in Objective-C to use the OSA infrastructure to control and communicate with OSA-compliant applications. With Scripting Bridge you can perform the same tasks in Objective-C that you can in AppleScript scripts.

Scripting Bridge is dynamic. At runtime it retrieves the scripting definition of a given application and generates Objective-C class implementations of the classes it finds in the scripting interface, including objects and methods representing properties, elements, commands, and so on. These objects become part of the Objective-C namespace that PyObjC and RubyCocoa scripts are bridged to, and through them these scripting languages are bridged to OSA-compliant applications. As a result, you can control and obtain data from those

applications from RubyCocoa and PyObjC code. And you also have at your disposal all the rich features and capabilities of the native languages, such as regular expressions, string manipulations, and easy access to the native libraries and modules.

To find out how to use Scripting Bridge in RubyCocoa and PyObjC scripts, see "Using Scripting Bridge in PyObjC and RubyCocoa Code" (page 43).

RubyOSA

RubyOSA is a open-source bridge that connects Ruby to the Apple Event Manager infrastructure, thereby enabling you to do in Ruby what you can do in AppleScript. It works by retrieving the scriptable definition of a given application (in its `sdef` file) and using that to populate a new namespace with classes, methods, constants, enumerations, and all other symbols described by the definition.

Most Mac OS X applications are scriptable, and they define their scriptable interface in the `sdef` XML format. RubyOSA parses this file and creates Ruby proxy objects with it on the fly. RubyOSA does its work transparently for you to build, send, and receive Apple events.

To give you an example of how simple and even elegant RubyOSA can be, consider the following code snippet, which gets the name of the current iTunes track:

```
require 'rbosa'
puts OSA.app('iTunes').current_track_name
```

RubyOSA is an improved alternative to RubyAEOSA. The latter bridge is implemented as a set of Ruby bindings to the Apple event C API, while RubyOSA is a higher level framework that completely hides the Apple event infrastructure. It is simpler and more efficient than RubyAEOSA.

You can download RubyOSA from http://rubyosa.rubyforge.org/ or, if you already have RubyGems installed, download and install it from the command line. To learn how, and for a practical look at RubyOSA, see "Using Scripting Bridge in PyObjC and RubyCocoa Code" (page 43).

py-appscript

py-appscript is an Python-OSA bridge that lets you control scriptable applications from Python scripts. It uses a high-level RPC mechanism for sending commands to applications via Apple events and converts data between common Python and Apple event types. py-appscript features an object-oriented style syntax and a simple embedded query language for identifying objects in an applicaitons object model.

You can download py-appscript from http://sourceforge.net/projects/appscript. The package includes installation instructions, examples, and documentation.

Multithreading With Ruby on Mac OS X

Because Ruby in its latest stable version (the 1.8 branch) is not thread-safe, you cannot call the Ruby runtime in a thread other than the main one. When Ruby is bridged to Objective-C this creates problems because Objective-C isn't able to call back to Ruby in a secondary thread. (If it did, an application would crash.) The version of RubyCocoa on Leopard consequently routes calls from Objective-C to Ruby so that all are on the main thread.

Ruby 1.8 also implements its threading model using the `setjmp` and `longjmp` primitives; this can sometimes cause unexpected behavior when a Ruby thread calls a Cocoa object, especially autorelease pools. Consequently, both the Ruby interpreter and the RubyCocoa bridge have been modified to properly handle these situations by saving and restoring the appropriate context variables during Ruby thread switching.

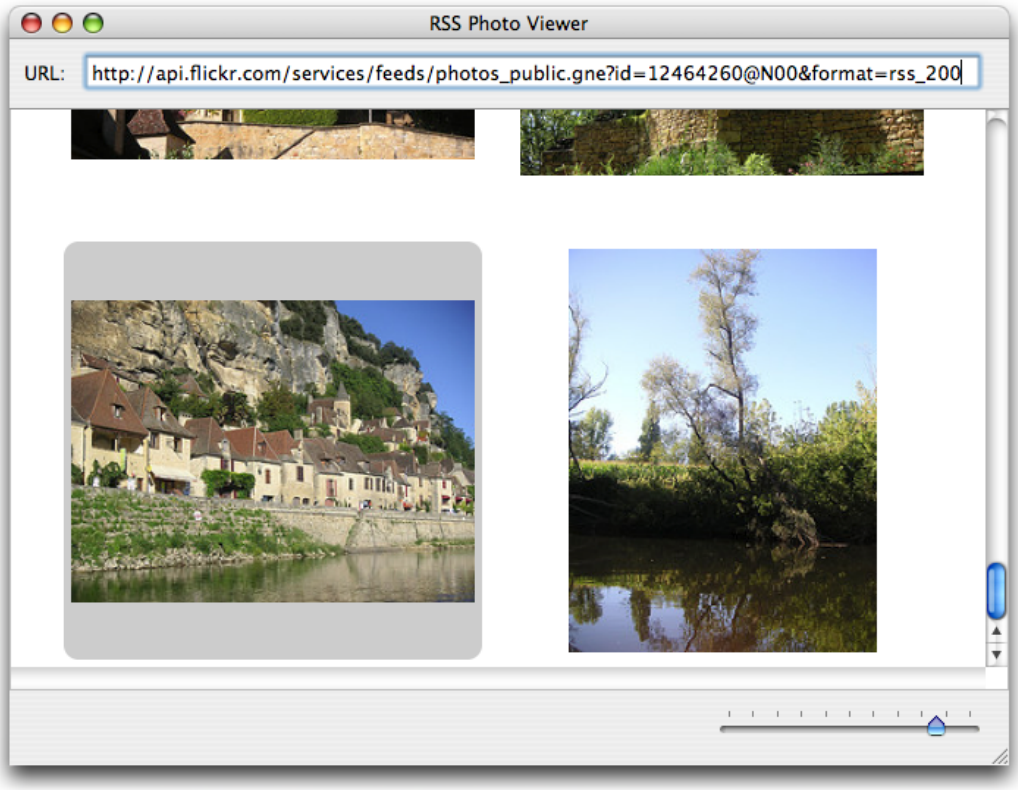
Fortunately, the next stable release of Ruby (the 2.0 branch) will be thread-safe and will use native threads. Unfortunately, this is not the version installed on Leopard.

Building a RubyCocoa Application: A Tutorial

This tutorial demonstrates how easy it is to create a RubyCocoa application using the developer applications Xcode and Interface Builder. It walks you through the steps for constructing the RSS Photo Viewer application, which is an example project installed in `<Xcode>/Examples/Ruby/RubyCocoa`. It assumes you have some knowledge of Ruby but not of RubyCocoa, and it assumes that you might be fairly new to the Mac OS X development environment.

When it is built and run, the RSS Photo Viewer (shown in Figure 1) lets you view photos that are accessed through a URL identifying an RSS feed. It enables you to scroll through the photos and zoom in on selected ones.

Figure 1 The RSS Photo Viewer application



By completing this tutorial you will gain familiarity with the following RubyCocoa development tasks on Mac OS X:

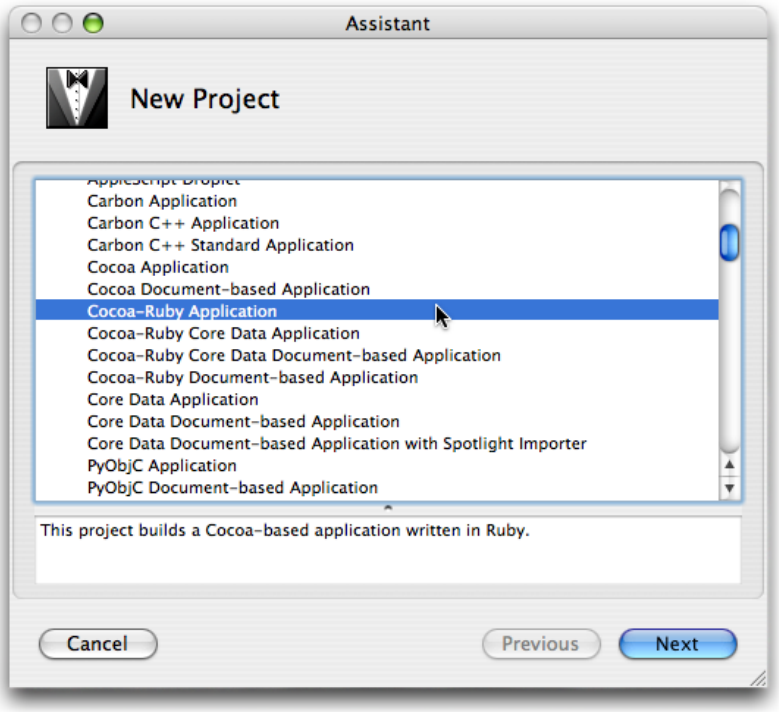
- Creating and setting up a RubyCocoa project
- Using Interface Builder to construct a user interface
- Defining a RubyCocoa class

- Defining and connecting outlets and actions
- Responding to delegation and data-source messages
- Implementing accessor methods
- Integrating a custom Ruby class

Creating and Configuring a RubyCocoa Project

There are several Xcode templates for RubyCocoa projects of various types: simple applications, document-based applications, Core Data applications, and Core Data document-based applications. The RSS Photo Viewer uses the simple RubyCocoa application template.

1. In Xcode, choose New Project from the File menu.
2. In the New Project assistant, select the Cocoa-Ruby Application template and click Next.

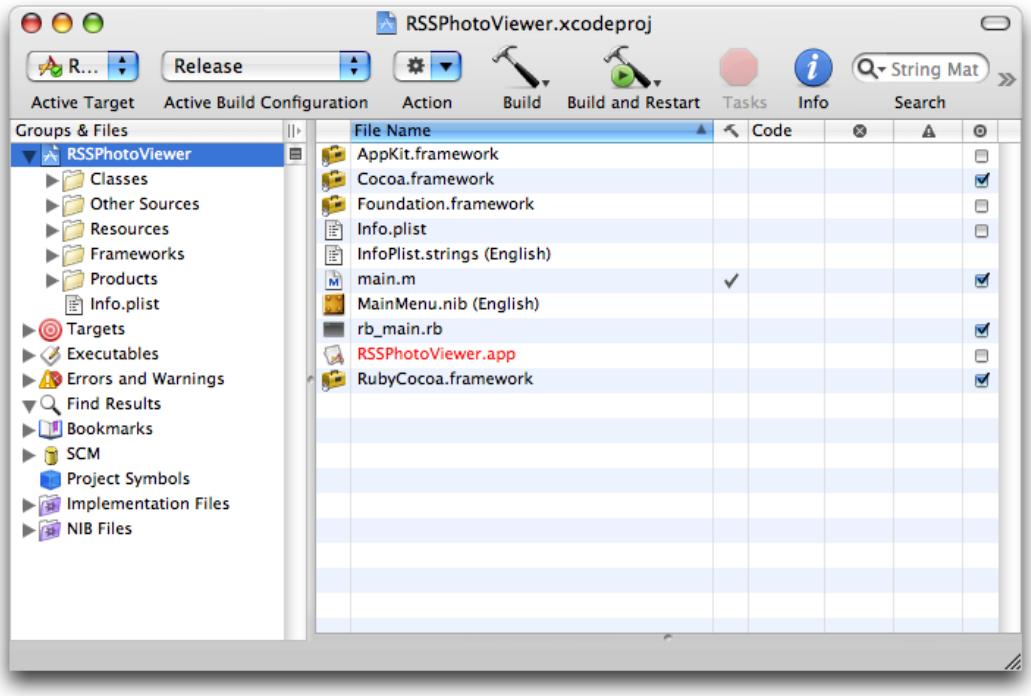


3. In the subsequent window, name the project folder "RSSPhotoViewer" and identify in a suitable location in the file system. Click Next.

Anatomy of a RubyCocoa Project

At first glance the project window looks like any other Cocoa project (see Figure 2). It has the necessary frameworks (including the RubyCocoa framework, a nib file with the main menu, the information property list (`Info.plist`), and the `main.m` file.

Figure 2 The initial project window for the RSS Photo Viewer project



However, it does have one file that you don't find in regular Cocoa application projects: `rb_main.rb`. When you build a RubyCocoa application, the Ruby scripts of the project are put in the `Resources` directory of the application bundle. If the application is double-clicked, the `rb_main.rb` script gets those Ruby and RubyCocoa files and loads them; if a RubyCocoa script is run from the command line (for debugging), however, the script instead calls the `NSApplicationMain` function.

Next add one line to the `rb_main.rb` script, the "`include OSX`" statement, as shown in Listing 1. By including the OSX module in the main scope, you are able to access RubyCocoa classes and methods directly—that is, without having to use the `OSX::` prefix.

Listing 1 The `rb_main.rb` script

```
require 'osx/cocoa'
# add the following line:
include OSX

def rb_main_init
  path = OSX::NSBundle.mainBundle.resourcePath.fileSystemRepresentation
  rbfiles = Dir.entries(path).select {|x| /\.rb\z/ =~ x}
  rbfiles -= [ File.basename(__FILE__) ]
  rbfiles.each do |path|
```

```
    require( File.basename(path) )
  end
end

if $0 == __FILE__ then
  rb_main_init
  OSX::NSApplicationMain(0, nil)
end
```

But what invokes the `rb_main.rb` script to begin with? Cocoa applications by default look for the execution entry point (that is, the function `main`) in `main.m`. However, this file in a RubyCocoa application project has different content than it does in a regular Cocoa application. As shown in Listing 2, the `main` function in RubyCocoa projects calls function `RBApplicationMain`, which takes as its first argument a string referencing `rb_main.rb`, and runs the script.

Listing 2 The `main.m` file in a RubyCocoa project

```
#import <RubyCocoa/RBRuntime.h>

int
main(int argc, const char* argv[])
{
  return RBApplicationMain("rb_main.rb", argc, argv);
}
```

Defining Classes, Targets, and Actions

The RSS Photo Viewer project has one significant source file, `RSSWindowController.rb`. This file contains definitions of two custom classes, a RubyCocoa subclass of `NSWindowController` and a simple custom Ruby subclass. We'll start with the `NSWindowController` subclass and define the outlets and actions that are used to control the user interface.

Note: An outlet is an archived connection between one object and another object (and is specified as an instance variable of one object). An action is a method invoked in an object (usually a custom object) called the target when another object such as a button or slider is manipulated; Interface Builder also archives the connection between the target and the other object (called a control). For more on these concepts, see ""Communicating With Objects"" in *Cocoa Fundamentals Guide*.

To add a RubyCocoa source file to the project, complete the following steps in the Xcode project:

1. Choose New File from the File menu.
2. Select "Ruby NSWindowController subclass" in the New File assistant window. Click Next.
3. In the subsequent assistant window name the new file "RSSRubyController.rb" and click Next.

   Xcode adds the file to the RSSPhotoViewer project.

The template file for the RubyCocoa subclass of `NSWindowController` includes some initial code: a `require 'osx/cocoa'` statement and an initial definition of the subclass. Change the name of this subclass from "RSSRubyController" (the name of the file) to "RSSWindowController":

```
class RSSWindowController < NSWindowController
end
```

Expand this initial class definition by typing the code shown in Listing 3.

Listing 3 Defining the outlet and actions of the RSSWindowController class

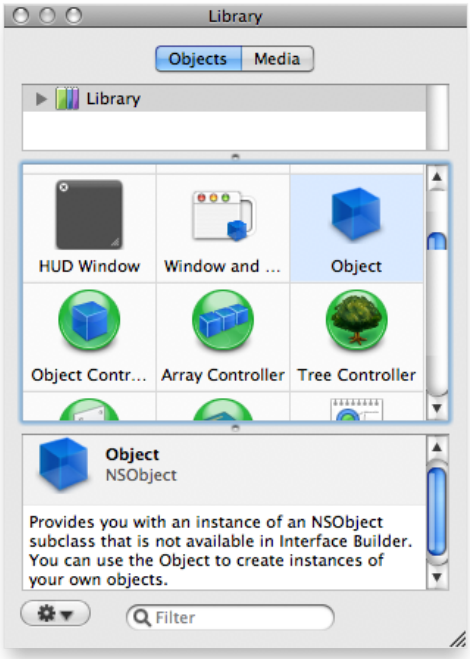
```
require 'osx/cocoa'

class RSSWindowController < NSWindowController
    ib_outlet :imageBrowserView

    # Actions

    def zoomChanged(sender)

    end
    ib_action :zoomChanged

    def parse(sender)

    end
    ib_action :parse

end
```

You just defined the outlet (`imageBrowserView`) and the two actions (`zoomChanged` and `parse`) that the `RSSWindowController` class uses for communicating with objects in the user interface. (In "Creating the User Interface" (page 25) you'll connect the outlet and actions to their user-interface objects.) To define an outlet use the `ib_outlet` method of the OSX module followed by the Ruby symbol ":imageBrowserView". You define an action by defining a method with a single argument named `sender`—the sender is the user-interface object sending the action message—followed by the `ib_action` method and the symbolized method name. Leave the action methods unimplemented for now; we'll return to them in "Implementing the Custom Window Controller" (page 31).

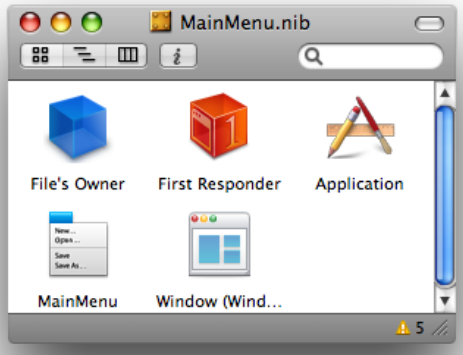
The next step is to import the `RSSWindowController` class, along with its action and outlet definitions, into the application's nib file.

1. Double-click the nib file `MainMenu.nib` in the Xcode project window to open it in Interface Builder.
2. Open the Library window (if it isn't displayed) by selecting Library from the Tools menu.

3. Locate the generic Object in the object library (that is, the browser in the Library window), either by browsing or search for it by typing “Object” in the window’s search field.



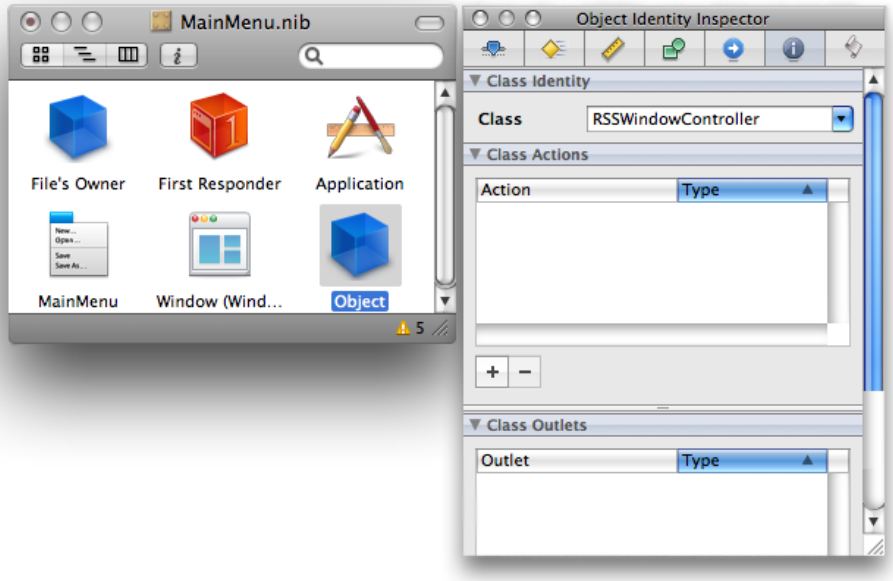
Drag this object into the nib file window, the window in this case with the title of MainMenu.nib.



The nib file window, which is sometimes called the nib document window, lets you examine the contents of a nib file. The default mode shows the top-level objects of the nib file—that is, those objects that are the top-level objects of an object graph (such as a window or menu) or that are standalone objects, such as controllers, which own no subordinate objects.

4. With the generic object selected, open the Identity pane of the inspector by choosing Identity Inspector from the Tools menu.

5. In the Class field, type “RSSWindowController” and press Return.



Completing the above steps imports the class into Interface Builder and assigns it as the class of the proxy object, which you can then use for target and action connections. This application is now “aware” of the `RSSWindowController` class, and automatically notices any future changes to the class—for example, additions of actions or removals of outlets.

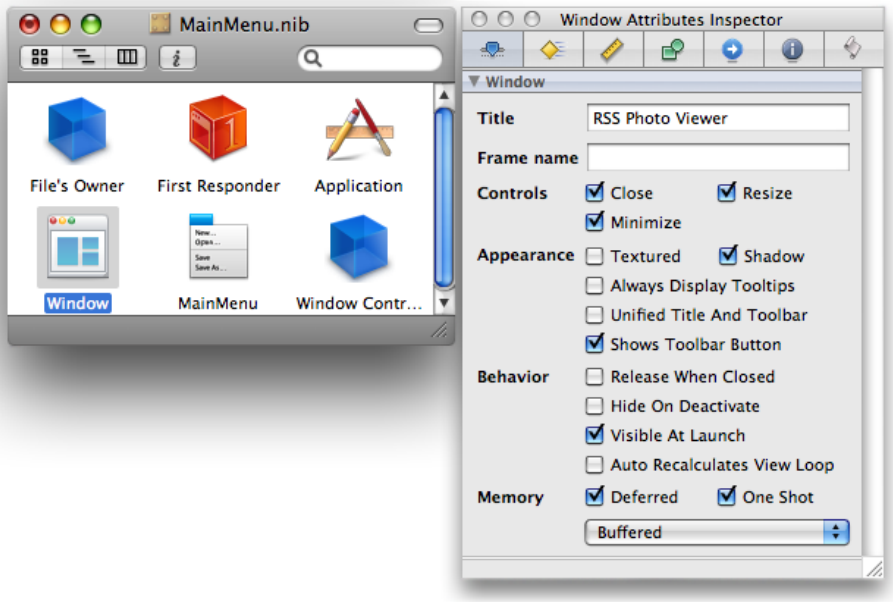
Creating the User Interface

The user interface of the RSS Photo Viewer application is simple. It is a single-window application, and on that window are only three objects:

- A labeled text field for specifying a URL
- A slider for zooming a selected image
- An Image Kit browser view for displaying the images

The Library window contains ready-made objects for the labeled text field and the slider. You drag these objects from the Library onto the window (provided by default) and then resize them, reposition them, and configure their other attributes. But before you do this, make sure the window is large enough to hold the objects. Using the image in Figure 1 (page 19) as a guide, resize the window by dragging the tab on the lower-right corner of the window.

Now that your attention is on the window, give it a title. Select its icon in the nib file window (if it isn’t already selected), and choose Attributes Inspector from the Tools menu to open the Attributes pane (or click the  button at the top of the inspector). Enter “RSS Photo Viewer” in the Title field.



Follow these steps to add and configure the URL text field:

1. In the object library find the Text Field object (you can search for it by typing “text field” in the Library window’s search field).
2. Drag this object (not Text Field Cell) and drop it on the upper part of the window.
3. Resize the text field using the resize handles on the edges of the object. (Make sure you leave space for the “URL” label”). Reposition it if necessary by dragging it over the window’s “surface.“
4. Locate the Label object in the object and drag it to a point left of the text field.

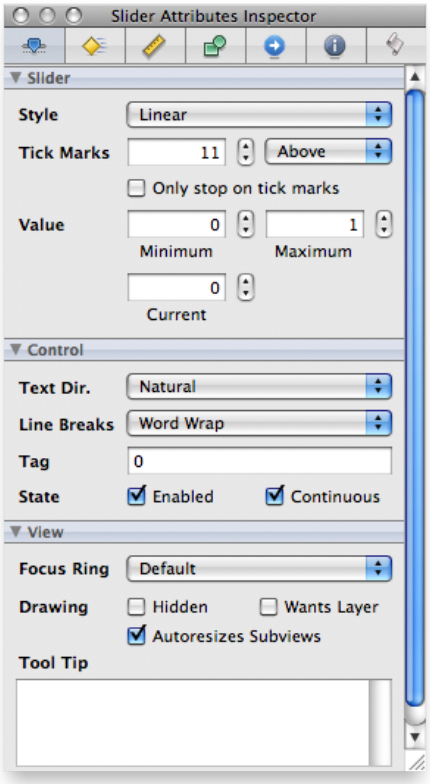
   This object is a text field too, but it is preconfigured to be read-only and to have a gray background.
5. Double-click “Label” and type “URL:” in place of the selected text. Resize the label if necessary and position it close to the text field.

You’ll notice as you move these objects around and resize them that blue lines appear. These are guide lines showing you how to place objects in relation to each other as recommended in the *Apple Human Interface Guidelines*.

Next find the horizontal slider in the object library and drag it to the lower part of the window. You will need to configure this object, following these steps:

1. Resize the object to about twice its default length.
2. Select the object and open the inspector to the Attributes pane (Tools > Attributes Inspector).)

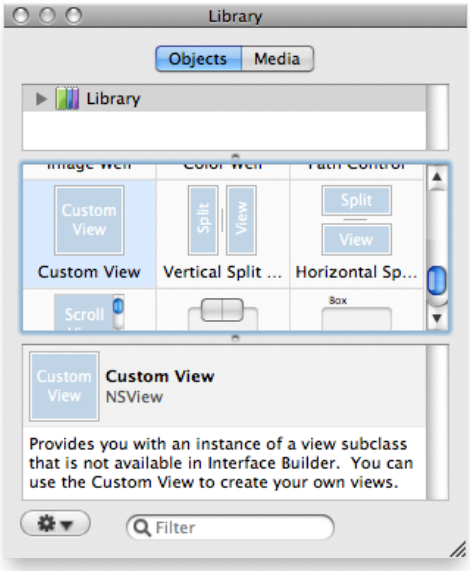
3. Set the number of tick marks and the minimum, maximum, and current values as shown in this example:



   Also make sure the Enabled and Continuous boxes in the Control slice are checked. Note that the inspector here is showing you the attributes belonging to the various inheritance layers of the slider: as an `NSSlider` object, as an `NSControl` object, and as an `NSView` object.

The final piece in the user interface of the RSS Photo Viewer application is the Image Kit image browser view (an `IKImageBrowserView` object). Because this Objective-C framework does not yet include an Interface Builder plug-in for its view objects, we will have to use the Custom View library object as a proxy, and then assign the `IKImageBrowserView` class to this proxy. Interface Builder is aware of this class, however, because the Application Kit framework has a dependency on the Quartz umbrella framework, which includes the Image Kit framework. The steps for setting up the image browser view are the following:

1. Find the Custom View object in the object library and drag it onto the window.

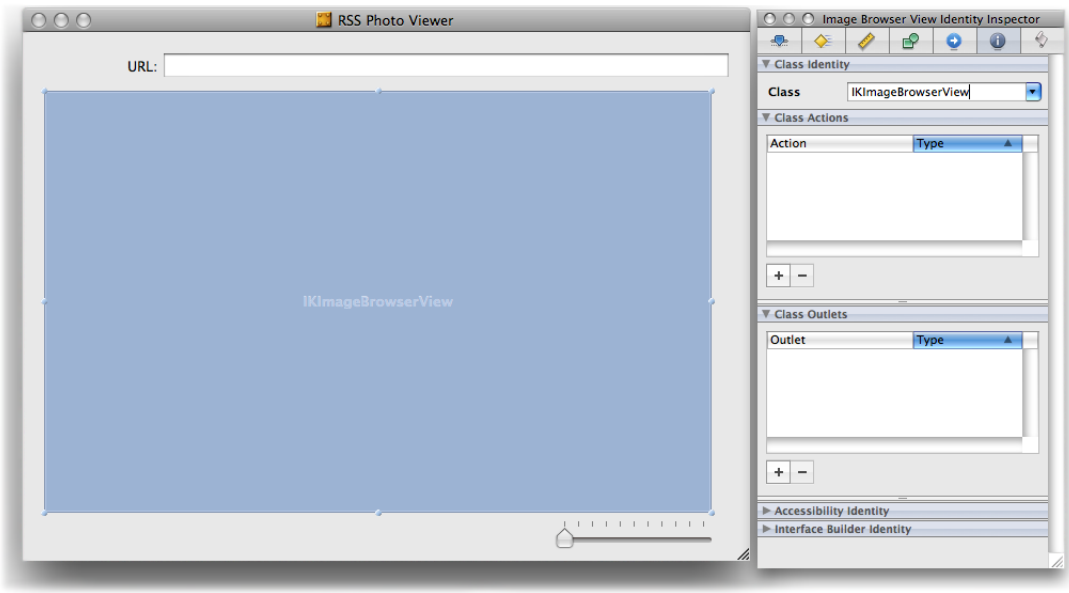


2. Resize the Custom View object to fill the space below the URL text field and above the slider.

   Note the blue guide lines for placement and resizing boundaries.

3. With the Custom View selected, open the Identity pane of the inspector.

4. Type “IKImageBrowserView” in the Class field and press Return.



Save the nib file. The RSS Photo Viewer application’s user interface is now complete. The next step is to hook up your outlet and action connections.

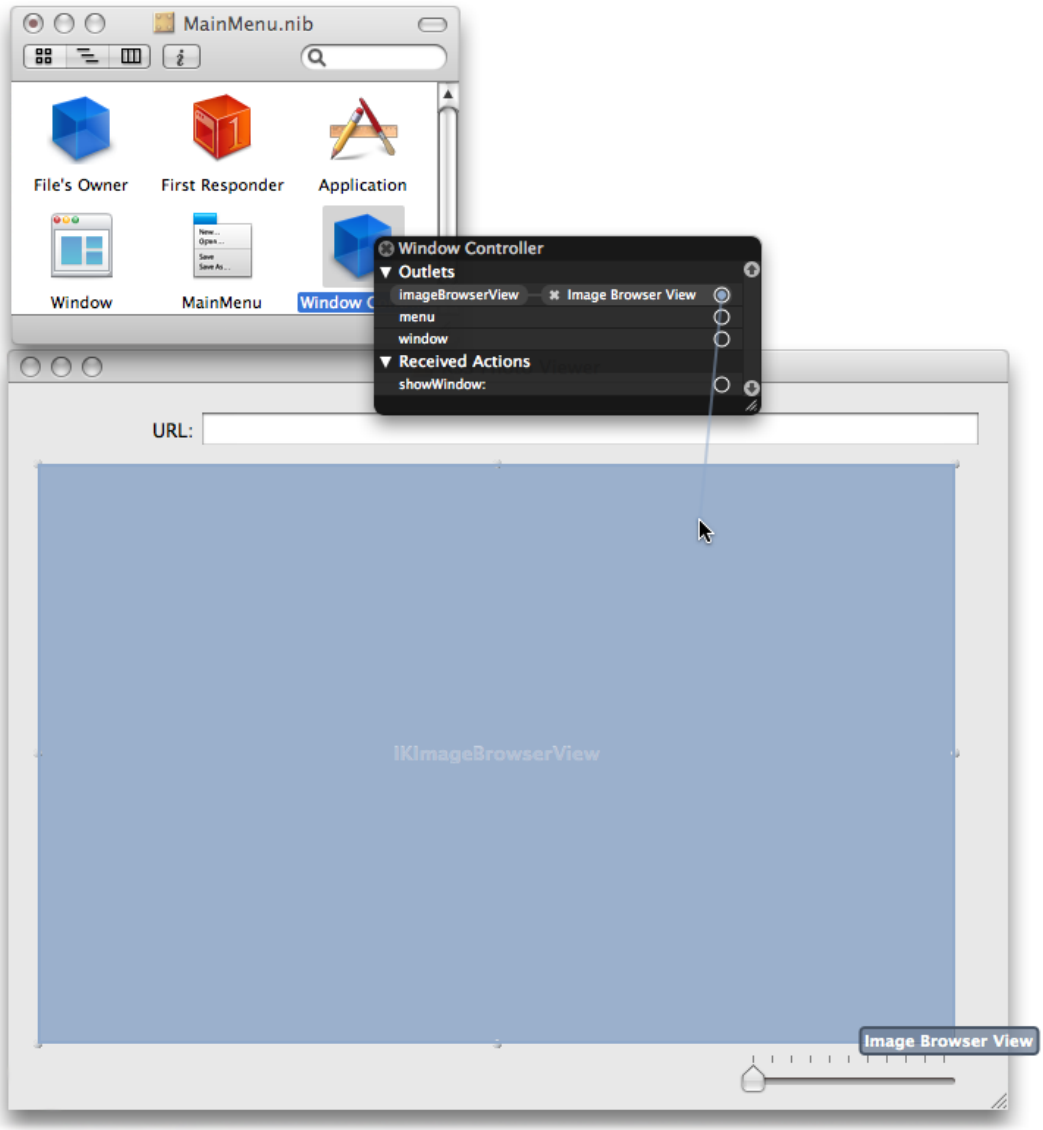
Connecting the Outlet and Actions

Before we leave Interface Builder and return to Xcode we need to connect the outlet and actions we defined in "Defining Classes, Targets, and Actions" (page 22) to their targets. Let's start with the outlet from the `RSSWindowController` class to the `IKImageBrowserView` object.

1. Select the `RSSWindowController` object in the nib file window and right-click (or Control-click) the mouse.

   The connections panel appears for that object, showing its possible connections.

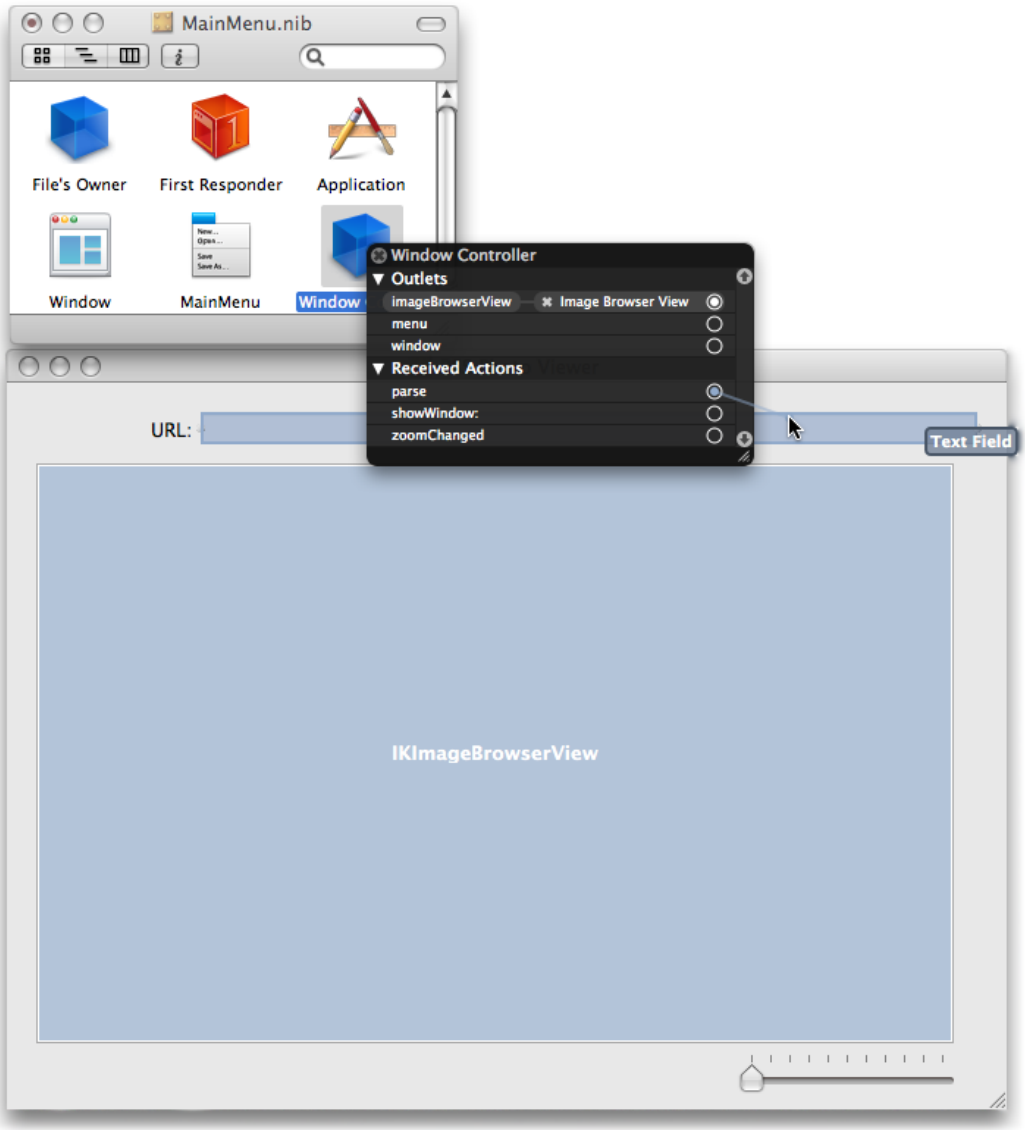
2. Click the mouse pointer in the circle next to the `imageBrowserView` outlet and drag a line to the `IKImageBrowserView` object.



3. Release the mouse button.

Next connect the action parse from the URL text field to the target object (`RSSWindowController`).

1. Select the object in the nib file window representing `RSSWindowController` and right-click (or Control-click) the mouse..
2. In the connections panel, drag a line from the circle next to parse in the Received Actions slice and drag it to the URL field.



3. Release the mouse button.

Complete the same sequence of steps for the slider object and the `zoomChanged` action. Then save the nib file and return to the Xcode project.

Implementing the Custom Window Controller

The implementation code of the RSS Photo Viewer application is centered around the programmatic interface of the `IKImageBrowserView` class of the Image Kit framework. The `RSSWindowController` class implements methods of informal protocols for data sources and delegates and calls `IKImageBrowserView` methods to set up and control the image browser. The RSSPhoto class, which you'll implement in "Implementing a Custom Ruby Class" (page 33) is a simple Ruby class that conforms to informal-protocol methods for objects that represent items in the image browser.

Note: The Image Kit framework was introduced in Mac OS X version 10.5 (Leopard).

Now it's time to write some code. We'll start by completing the implementation of the `RSSWindowController` class. Open `RSSRubyController.rb` in Xcode and add the following method:

```
def awakeFromNib
   @cache = []
   @imageBrowserView.setAnimates(true)
   @imageBrowserView.setDataSource(self)
   @imageBrowserView.setDelegate(self)
 end
```

The Application Kit framework sends the `awakeFromNib` message to all interested objects when all nib-file objects have been unarchived and loaded into memory. This method presents an opportunity for controller objects to perform initializations involving objects unarchived from nib files (view objects). The `awakeFromNib` method of `RSSWindowController` sets the `animates` property of the `IKImageBrowserView` object and assigns itself as the delegate and data source of that object. It also initializes a `@cache` instance variable, a Ruby Array object that contains the current collection of RSSPhoto objects (representing photos).

Next insert two more `require` statements after the `require 'osx/cocoa'` statement.

```
require 'rss'
require 'open-uri'
```

As you'll soon see, the `RSSWindowController` class uses the open-uri library for accessing HTTP resources. It also uses the RSS library for accessing information disseminated on the Internet through the RSS protocol.

Implement the `zoomChanged` and `parse` action methods as shown in Listing 4.:

Listing 4 Implementing the action methods

```
def zoomChanged(sender)
    @imageBrowserView.setZoomValue(sender.floatValue)
  end

  def parse(sender)
    begin
      uri = URI.parse(sender.stringValue.to_s)
      raise "Invalid URL" unless uri.respond_to?(:read)
      @parser = RSS::Parser.parse(uri.read, false)
      @cache.clear
      @imageBrowserView.reloadData
    rescue => e
      NSRunAlertPanel("Can't parse URL", e.message, 'OK', nil, nil)
```

```
    end
  end
```

There are a few things to note about this code:

- The `zoomChanged` message is sent when the user moves the slider under the image browser; the implementation here gets the new float value from the `sender` of the message (the slider) and sets the zoom factor of the image browser to that value.
- In implementing its `parse` action method, the `RSSWindowController` class uses the `parse` method of the open-uri library's URI class to validate the URL fetched from the text field (via the `sender.stringValue` call) and create a URI object from it. If the URL is not valid, it raises an exception.
- It then uses the `parse` method of the RSS library to parse the RSS stream referenced by the URI object and store the individual RSS entries in a Parser object referenced by the `@parser` instance variable. It then clears the local cache of photos and tells the image browser to reload its data.
- If any of the foregoing steps raises an exception, the parse method of the `RSSWindowController` class handles it by displaying an alert panel using the Application Kit's `NSRunAlertPanel` function.

When the `IKImageBrowserView` object receives a `reloadData`message, it requests from its data source—in this case, the `RSSWindowController` object—the images to display by sending it first a `numberOfItemsInImageBrowser`. Depending on the number of items returned by this method (assuming it's a positive value), the `IKImageBrowserView` object then sends one or more `imageBrowser:itemAtIndex:` messages to its data source. Listing 5 shows how `RSSWindowController` implements the methods invoked by these messages.

Listing 5 Implementing the data-source and delegate methods

```
def numberOfItemsInImageBrowser(browser)
  @parser ? @parser.items.length : 0
end

def imageBrowser_itemAtIndex(browser, index)
  photo = @cache[index]
  if photo.nil?
    item = @parser.items[index]
    url = item.description.scan(/img src="([^"]+)/).first.first
    photo = RSSPhoto.new(url)
    @cache[index] = photo
  end
  return photo
end

def imageBrowser_cellWasDoubleClickedAtIndex(browser, index)
  NSWorkspace.sharedWorkspace.openURL(@cache[index].url)
end
```

Let's examine the interesting aspects of this code, especially the RubyCocoa parts. The `imageBrowser_itemAtIndex` and `imageBrowser_cellWasDoubleClickedAtIndex` methods illustrate the RubyCocoa convention of replacing all keyword colons of Objective-C method signatures with underscores, except for the final colon. The implementation of `imageBrowser_itemAtIndex` checks if the photo referenced by the index value (of the browser) is in its cache of photos; it the photo doesn't exist, it gets the corresponding URL from the RSS Parser object and with that creates a `RSSPhoto` object, adds it to the cache, and returns it to the image browser, which displays the photo. (You will implement the `RSSPhoto` class in the following section, "Implementing a Custom Ruby Class.")

Recall that in `awakeFromNib` you set the `RSSWindowController` object to be the delegate of the `IKImageBrowserView` object. The image browser sends the `imageBrowser_cellWasDoubleClicked` message to its delegate when a user clicks on an image. This implementation uses the `NSWorkspace` method `openURL:` to open the image in the preferred application.

Implementing a Custom Ruby Class

The RSSPhoto class is a simple Ruby subclass in that it implicitly inherits from the root Ruby class. Although it doesn't inherit from a Cocoa class, as does RSSWindowController, it does implement the required methods of the `IKImageBrowserItem Protocol` informal protocol. By doing so, it represents an image that can be displayed in the `IKImageBrowserView` object. .

Listing 6 shows the RubyCocoa code used in implementing the RSSPhoto class, which is included in the `RSSRubyController.rb` file. The `imageUID`, `imageRepesentationType`, and `imageRepresentation` method implementations are required by the `IKImageBrowserItem` protocol. The `imageRepresentation` class provides the image browser with the `NSImage` object to display, using the `initByReferencingURL:` initializer; note how this method uses Ruby syntax to lazily create the image object.

Listing 6 Implementation of the RSSPhoto class

```
class RSSPhoto
  attr_reader :url

  def initialize(url)
    @urlString = url
    @url = NSURL.alloc.initWithString(url)
  end

  # IKImageBrowserItem protocol conformance

  def imageUID
    @urlString
  end

  def imageRepresentationType
    :IKImageBrowserNSImageRepresentationType
  end

  def imageRepresentation
    @image ||= NSImage.alloc.initByReferencingURL(@url)
  end
end
```

Using RubyOSA

RubyOSA is a bridge that lets developers control scriptable applications, including the Finder, using the Ruby scripting language. An application is called scriptable when it makes its operations and data available in response to messages called Apple events. RubyOSA provides a bridge between Ruby and the Open Scripting Architecture (OSA), an infrastructure for interprocess communication that uses Apple events as its mechanism for event dispatching and data transport. (AppleScript is the original OSA scripting language, and is still quite popular.)

A scriptable application specifies the set of scripting terms it understands and its scriptable interface in an XML dictionary called an `sdef` file ("sdef" for scriptable definition). At runtime RubyOSA parses the scriptable definition of a given application and populates a new namespace with classes, methods, constants, enumerations, and all other symbols described by the definition. It also dynamically creates Ruby proxy objects to represent these symbols and uses OSA mechanisms to build and send Apple events to applications and receive their responses.

RubyOSA has some obvious advantages, especially for Ruby programmers. With it you can control applications on Mac OS X and get requested objects back from them. You can do anything with these object that you can do in regular Ruby code, such as string manipulations and regular expressions. Your code also has access to all installed Ruby modules and libraries. Finally, you can combine RubyOSA and RubyCocoa in the same script to apply the technologies of the Mac OS X frameworks to the access to scriptable applications that OSA makes possible.

Installing RubyOSA

You can download the latest version of RubyOSA from its open-source repository and install it on your system by running the following command in a Terminal shell:

```
sudo gem install rubyosa
```

The Basics

The essential idea behind using RubyOSA is to get a proxy instance of a scriptable application and then send messages to it. The messages that you can send are described in the application's scriptable definition, or dictionary. Let's start by looking at a simple example (Listing 1).

Listing 1 The `iTunes_inspect.rb` script

```
# Quick inspection of iTunes' sources, playlists and tracks.

require 'rubygems'
require 'rbosa'
```

```
app = OSA.app('iTunes')
OSA.utf8_strings = true
app.sources.each do |source|
    puts source.name
    source.playlists.each do |playlist|
        puts " -> #{playlist.name}"
        playlist.tracks.each do |track|
            puts "     -> #{track.name}" if track.enabled?
        end
    end
end
```

When you run this script from the command line, it prints information similar to the following lines:

```
Library
-> Classical CD
     -> Toccata & Fugue in D Minor
     -> Air on the G String (2nd movement from Orchestral Suite No. 3 in D)
     -> No.13 Waltz of the Flowers
     -> Montagues And Capulets
     -> Egmont Overture, Op 84
     -> Die Zauberflöte
     -> Horn concerto 3EFlat, 1. Allegro
     -> Horn concerto 3EFlat 2. Romance. Larguetto
     -> Horn concerto 3EFlat, 3. Allegro
     ........
```

The first thing to notice about the script in Listing 1 (page 43) is `require 'rbosa'`. This statement loads the rbosa library, which includes the OSA class. The next line of the script is equally important:

```
app = OSA.app('iTunes')
```

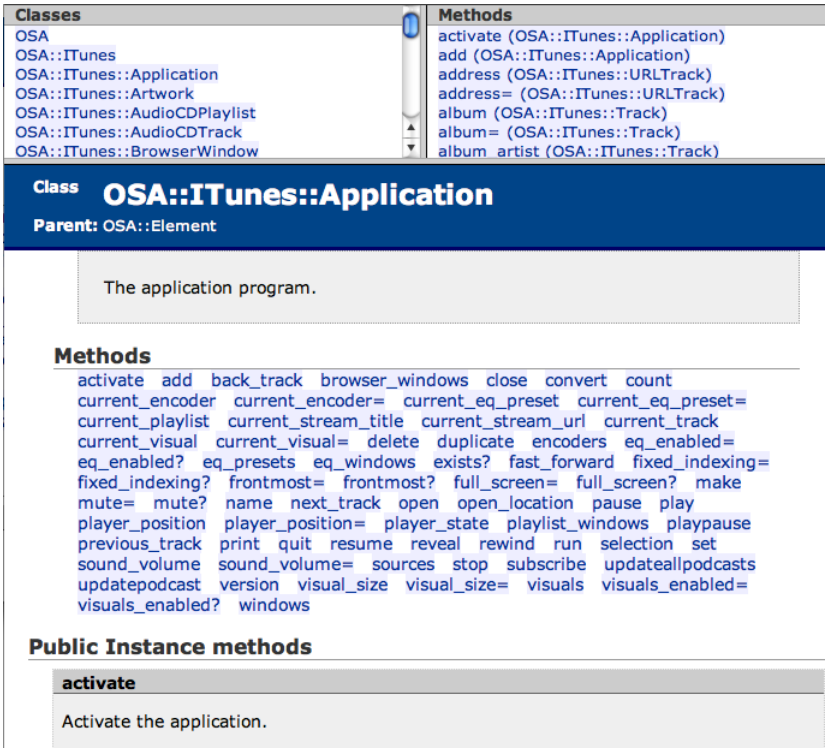
This line returns a proxy Ruby object representing a scriptable application, in this case iTunes. (Note that all you have to do specify the name of the application; you don't have to include its file-system location or its extension.) From this point on, the script sends messages to the application object and the objects it "contains," and performs Ruby operations on the results. In RubyOSA's internal representation of a scriptable application, a hierarchy of objects descends from the application object; sending a message to the application object may return a collection objects, each of which may be a collection of subordinate objects. You can send appropriate messages to each of these objects. Take these lines as an example:

```
app.sources.each do |source|
    puts source.name
    source.playlists.each do |playlist|
        puts " -> #{playlist.name}"
```

The `sources` message to the iTunes proxy object returns an object that implements the Ruby Array interface. The script then loops through the array and in a block sends a `name` message to each fetched object (`source`, representing a music source) and prints the returned Ruby string. It next sends `playlists` to `source` and iterates through the array returned from that call, which represents the playlists associated with that music source. It prints the name of each playlist. And so on proceeds the script.

This might seem simple and straightforward—and it is—but a question might arise: where do you find out which messages you can send to a scriptable application's hierarchy of objects? RubyOSA includes a documentation tool, `rdoc-osa`. Using this you tool you can generate a set of HTML pages that document the scriptable definition of a Mac OS X application. "The Basics" shows the opening page of the iTunes documentation.

Figure 1 A page from the rdoc-osa documentation for iTunes



If you were to use this documentation, you would find that sending `sources` to a proxy object representing the iTunes application returns an array (or list) of OSA::iTunes::Source objects. Sending `playlists` to one of these objects returns an array of OSA::ITunes::Playlist objects. And sending `tracks` to one of these objects returns an array of OSA::ITunes::Track objects. You can then send `name` to one of these objects to get the name of the track.

The OSA Class

You might have wondered about the following line in the sample script in Listing 1 (page 43):

```
OSA.utf8_strings = true
```

OSA is a Ruby class in its own right, and has other methods besides `app`, among them `utf8_strings`. Listing 2 describes the methods of the OSA class.

Table 1 Methods of the OSA class

Method	Description
`OSA.app(`*application-specifier*`)`	Returns an OSA proxy object representing the application specified by the string *application-specifier*. You can specify the application by name, by bundle ID, by path, or by signature. For more information on specifying applications, both local and remote, see below.

Method	Description
`OSA.lazy_events`	Controls whether OSA proxy objects are resolved on demand or are resolved automatically. By default objects are resolved on demand (`true`), meaning that OSA objects are resolved only when necessary. Object resolution involves the sending of an Apple event to discover the type of an object. Thus automatic resolution can have performance implications when there is a considerable number of objects (for example, a loop to get all iTunes tracks). However, this might be unavoidable when the target application's scriptable definition doesn't describe the types of objects, instead using the "reference" type for each of them.
`OSA.utf8_strings`	Controls whether strings will be encoded as Unicode (UTF8) or as ASCII. By default this property is set to `false` because some applications might not be able to handle Unicode strings.
`OSA.timeout`	Controls the timeout period for getting responses to Apple events. The value is expressed in ticks (seconds). By default it's set to -1, which is about one minute. A value of -2 means there is no timeout.
`OSA.wait_reply`	Controls whether RubyOSA should expect a result from the Apple events it sends. If set to `nil` (the default), RubyOSA determines the value by examining the scriptable definition; this might (rarely) result in a malformed application command. Set this value to `true` (or `false`) to force RubyOSA to send back (or not send back) a return value.

All RubyOSA objects inherit from the `OSA::Element` class, which is completely opaque to the user.

With the RubyOSA `app` method you can identify scriptable applications in several ways:

- By name, simply by putting the application name (minus the app extension) between single quotation marks.

  Example: `OSA.app('Finder')`

  This simple style of argument is a convenience for `:name => 'AppName'`. RubyOSA uses Launch Services to locate the scriptable application to launch and use.

- By file-system path, using the `:path` key.

  Example: `OSA.app(:path => '/Users/jdoe/Applications/BBEdit.app')`

- By the application's bundle ID, using the `:bundle_id` key.

  Example: `OSA.app(:bundle_id => 'com.apple.iTunes')`

- By an application's four-character creator signature (if any), using the `:signature` key.

  Example: `OSA.app(:signature -> 'woof')`

The `app` method also lets you specify applications on remote machines as well as locally—thus you can control and get data from applications that aren't even installed on your local system. After specifying the application by name, you add one to three key-value pairs identifying the machine, the user name, and the password. For each pair, use the `:machine`, `:username`, and `:password` keys, respectively. For example:

```
OSA.app('iTunes', :machine => 'kubla.acme.com', :username => 'jdoe' :password
=> '3x534C2')
```

There are a few things to be aware of when calling the `app` method to get proxy instances of remote applications: First, you may only specify the remote-access key-value pairs when the first argument specifies the application by name. Second, if you omit the `:username` or `:password` keys (or both), RubyOSA prompts for the user name and password (or both).

- The Remote Apple Events checkbox in the Sharing pane of System Preferences on the remote machine should be checked for your RubyOSA script to control its applications.
- You may only specify the remote-access key-value pairs when the first argument specifies the application by name.
- if you omit the `:username` or `:password` keys (or both), RubyOSA prompts for the user name and password (or both).

Conversions and Conventions

When you send a message whose name has a plural form (for example, `sources`), what you get in return may look and behave like an Array, but it is actually an list element (OSA::ObjectSpecifierList) containing object specifiers—that is, references to real objects. Although the Ruby Array class is not directly used in this case, the OSA::ObjectSpecifierList class conforms to the Array interface; in other words, it mixes the Enumerable module. Therefore you can call *most* of the methods on an object-specifier list that you can call on an Array.

Methods with names such as `title` and `name` refer to properties in a scriptable definition and return the appropriate Ruby objects (in both these cases, String objects). On the other hand, methods such as `current_track` return an object specifier, in this case an object specifier of the OSA::ITunes::Track class. The rule that RubyOSA follows to distinguish between these two general types of properties is that when the type of the property is defined within the target application's scriptable definition (as `current_track` is), it returns an object specifier. Otherwise it assumes the object is of a primitive type (`String`, `Integer`, `Date`, and so on) and it resolves the return value directly by querying for the type with an extra Apple event.

Some Examples

To better appreciate the varieties of ways in which you might use RubyOSA, let's examine a few of the examples installed in `/Developer/Examples/Ruby/RubyOSA`. The script in Listing 3 creates a proxy instance of the Finder application and from it requests the current contents of the Desktop. Using Ruby regular expressions and string-manipulation methods, it formats and prints these items.

Listing 2 The `Finder_show_desktop.rb` script

```
# Lists the content of the Finder desktop.

require 'rubygems';
require 'rbosa'

ary = OSA.app('Finder').desktop.entire_contents.get
ary.each do |x|
    next unless x.is_a?(OSA::Finder::Item)
    puts "#{x.class.name.sub(/^.+::/, '').sub(/_/, ' ').ljust(25)} #{x.name}"
end
```

Listing 3 is a script that displays the album artwork associated with the iTunes track that is currently playing. Note that it creates a temporary file to hold the image data and then makes a `system` call to open this file in the Preview application. With the `system` call your script can do anything that can be done at the command line.

Listing 3 The `iTunes_artwork.rb` script

```
# Open the artwork of the current iTunes track in Preview.

require 'rubygems'
require 'rbosa'

artworks = OSA.app('iTunes').current_track.artworks
if artworks.size == 0
  puts "No artwork for current track."
  exit 1
end

fname = '/tmp/foo.' + artworks[0].format.downcase.strip
File.open(fname, 'w') { |io| io.write(artworks[0].data) }
system("open -a Preview #{fname}")
```

What is noteworthy about the script in Listing 4 is that it exchanges data between proxy instances of two applications, TextEdit and Mail. It gets the selected messages in all current Mail viewers and copies each the content of each message to a TextEdit window.

Listing 4 The `get_selected_mail.rb` script

```
# Copy contents of selected Mail messages to a TextEdit window

require 'rubygems'
require 'rbosa'

textedit = OSA.app('TextEdit')
mailApp = OSA.app('Mail')
viewers = mailApp.message_viewers
viewers.each do |viewer|
    viewer.selected_messages.each do |message|
        textedit.make(OSA::TextEdit::Document).text = message.content
     end
end
```

Finally. the Listing 5 script updates in the iChat status area the time the system has been running since it was last booted. It is similar to Listing 1 (page 43) it that it makes a system call, but instead of calling the `system` method, it invokes the `uptime` command simply by enclosing it single quotes. It then formats the output of the command and assigns this formatted string to the iChat `status_message` property. All this occurs in a closed loop, which is re-executed after a five-second pause, which causes a periodic update of the system-uptime message.

Listing 5 The `iChat_uptime.rb` script

```
# Periodically set your iChat status to the output of uptime(1).

require 'rubygems'
require 'rbosa'
```

```
app = OSA.app('iChat')
previous_status_message = app.status_message
trap('INT') { app.status_message = previous_status_message; exit 0 }
while true
    u = `uptime`
    hours = u.scan(/^\s*(\d+:\d+)\s/).to_s + ' hours'
    days = u.scan(/\d+\sdays/).to_s
    app.status_message = "OSX up #{days} #{hours}"
    sleep 5
end
```

This script traps interruption of the script (such as happens when the user presses Control-C) and restores the previous value of the iChat status message before exiting.

Documenting Application Dictionaries

You can use the `rdoc-osa` tool to generate HTML or `ri` documentation for the dictionary (that is, scriptable definition) of an application. Using `rdoc-osa` is simple. For example, to generate HTML documentation of the iTunes dictionary, you would enter the following command on a shell's command line:

```
rdoc-osa --name iTunes
```

The `ruby-osa` tool generates the documentation from the application's dictionary and puts in in a folder named `doc` in the current working directory. Instead of identifying the application by name, you can identify it by path, bundle ID, or four-character creator signature. To generate `ri` documentation instead of HTML, append "`--ri`" to the command.

Note: `ri` is a Ruby tool for viewing documentation in a format familiar to Ruby programmers. To learn more about `ri`, type "`ri --h`" at the command line.

To get help on `rdoc-osa`, enter "`rdoc-osa --h`" at the command line. The `rdoc-osa` tool accepts all options used in `rdoc`, the documentation generator for Ruby classes and modules. Enter "`rdoc --h`" at the command line to learn about the options for that tool.

Using Scripting Bridge in PyObjC and RubyCocoa Code

Scripting Bridge is a technology that you can use in PyObjC and RubyCocoa scripts to communicate with scriptable applications—that is, applications with scripting interfaces compliant with the Open Scripting Architecture (OSA). With Scripting Bridge, RubyCocoa and PyObjC scripts can do what AppleScript scripts can do: control scriptable applications and exchange data with them. The Scripting Bridge framework implements a bridge between OSA and the Objective-C runtime. It reads the scripting definition of applications and dynamically populates the Objective-C namespace with objects and methods representing the various items it finds (scripting objects, elements, commands, properties, and so on). RubyCocoa and PyObjC are also bridges to the Objective-C runtime and thus have access to everything in a program's namespace, including Scripting Bridge–created objects.

The section on Scripting Bridge in "Ruby and Python on Mac OS X" (page 9) surveys the technology, describing its capabilities and architecture. The following sections describe how your RubyCocoa and PyObjC scripts and programs can take advantage of Scripting Bridge.

Important: The Ruby and Python bridges to Objective-C use framework metadata to learn about non-introspectable types. However, the classes that Scripting Bridge dynamically generates currently do not include any metadata. This mismatch leads to some limitations when using Scripting Bridge in RubyCocoa and PyObjC code. For example, Scripting Bridge declares enumerated types in its Objective-C headers, but these are not bridged to RubyCocoa and PyObjC as symbolic entities. To use an enumerated value in a script you must specify the corresponding integral value; for instance, the value of iChat's 'away' enumerator would be 0x61776179. Another important mismatch to be aware of are Boolean parameters and return types. Scripting Bridge declares these as `BOOL`, which is a `typedef` for `signed char`. However, Ruby evaluates all numbers, even zero, as logically true because they are Number objects. Consequently when using Scripting Bridge in RubyCocoa you must test Boolean values for equality to zero and not whether they are logically false.

These limitations apply to the initial version of Scripting Bridge, which was introduced in Mac OS X version 10.5.

The Basics

The essential idea behind using Scripting Bridge is to get an object representing a scriptable application and then send messages to that object. Messages can result in objects being returned from the application object, and you can send messages to those objects—and so on down the object graph. The messages that you can send are described in the application's scripting interface, or dictionary. Let's start by looking at a simple example using RubyCocoa (Listing 1).

Listing 1 The `iTunes_inspect.rb` script

```
require 'osx/cocoa'
include OSX
OSX.require_framework 'ScriptingBridge'
```

```
iTunes = SBApplication.applicationWithBundleIdentifier_("com.apple.iTunes")
iTunes.sources.each do |source|
  puts source.name
  source.playlists.each do |playlist|
    puts " -> #{playlist.name}"
    playlist.tracks.each do |track|
      puts "      -> #{track.name}" if track.enabled?
    end
  end
end
```

When you run this script from the command line, it prints information similar to the following lines:

```
Library
-> Classical CD
     -> Toccata & Fugue in D Minor
     -> Air on the G String (2nd movement from Orchestral Suite No. 3 in D)
     -> No.13 Waltz of the Flowers
     -> Montagues And Capulets
     -> Egmont Overture, Op 84
     -> Die Zauberflöte
     -> Horn concerto 3EFlat, 1. Allegro
     -> Horn concerto 3EFlat 2. Romance. Larguetto
     -> Horn concerto 3EFlat, 3. Allegro
     ........
```

The first thing to notice about the script in Listing 1 (page 43) are the first three lines, which set up the necessary environment. The first statement loads the osx/cocoa library, and the next two statements append the features of the OSX module and import the Scripting Bridge framework from it.. All three statements in the given order are required for RubyCocoa programs that use the Scripting Bridge.

The next line is particularly interesting:

```
iTunes = SBApplication.applicationWithBundleIdentifier_("com.apple.iTunes")
```

This statement is a message expression that returns an proxy Ruby object representing a scriptable application, in this case iTunes. The message invokes the class method `applicationWithBundleIdentifier:` of the `SBApplication` class of the Scripting Bridge framework. This method requires that you identify the scriptable application by its bundle identifier. (See "The Scripting Bridge Classes" (page 45) for more about `SBApplication` and its methods for creating application objects.)

From this point on, the script sends messages across the bridge to the scriptable-application object and the objects it contains, gets the values of certain properties, and performs Ruby operations on the results. In Scripting Bridge's internal representation of a scriptable application, a hierarchy of objects descends from the application object; sending a message to the application object may return elements, which are collections of other objects; each object in the element array may have elements containing objects, and so on. You can send appropriate messages to each of these objects. Take these lines as an example:

```
iTunes.sources.each do |source|
    puts source.name
    source.playlists.each do |playlist|
        puts " -> #{playlist.name}"
```

The `sources` message to the iTunes proxy object returns an object that implements the Ruby Array interface; on the other side of the bridge, this is an `SBElementArray` object. The script then loops through the array and in a block sends a `name` message to each fetched object (`source`, representing a music source) and

prints the returned Ruby string. It next sends `playlists` to `source` and iterates through the array returned from that call, which represents the playlists associated with that music source. It prints the name of each playlist. And so on until it gets and prints the name of each track.

Using Scripting Bridge in a PyObjC script is as simple and straightforward as it is for RubyCocoa. Here is a short script that prints (to standard output) the name of the track currently playing on iTunes:

```
from Foundation import *
from ScriptingBridge import *

iTunes = SBApplication.applicationWithBundleIdentifier_("com.apple.iTunes")
print iTunes.currentTrack().name()
```

In this case, setting up the environment for using Scripting Bridge involves just two import statements, one for the Foundation framework and the other for the Scripting Bridge framework.

The Scripting Bridge also allows you to add objects to a scriptable application. For this it declares the following `SBApplication` method, which returns the class object for the scripting class specified in the receiver's scripting definition:

```
+ (Class)classForScriptingClass:(NSString *)className;
```

Once you have the `Class` object, you can instantiate a scripting object of the indicated type and add it to the application. If, for example, you wanted to add a playlist to iTunes, in PyObjC code you could similar to the example in Listing 2 (page 45):

Listing 2 Adding an object to a scriptable application in PyObjC code

```
from Foundation import *
from ScriptingBridge import *

iTunes = SBApplication.applicationWithBundleIdentifier_("com.apple.iTunes")
p = {'name':'Testing'}
playlist =
iTunes.classForScriptingClass_("playlist").alloc().initWithProperties_(p)
iTunes.sources()[0].playlists().insertObject_atIndex_(playlist, 0)
```

Scripting Bridge does not actually create an object in the target application until you add the allocated and initialized object to an appropriate element array (`SBElementArray`), such as `playlists` in the above example.

The Scripting Bridge Classes

The Scripting Bridge framework has three public Objective-C classes:

- `SBApplication`—A class whose objects represent scriptable applications
- `SBElementArray`—A class whose objects represent collections of elements in the scripting definition
- `SBObject`—The base class of scripting objects in a scriptable application

The class factory methods of `SBApplication` enable you to obtain an object representing an OSA-compliant application. The following methods return an `SBApplication` object representing an application (autoreleased in memory-managed environments):

`+ (id) applicationWithBundleIdentifier:(NSString *)bundleID;`
Finds the application using its bundle identifier, for example `@"com.apple.iTunes"`. This is the recommended approach for most situations, especially when the application is local, because it dynamically locates the application.

`+ (id) applicationWithURL:(NSURL *)url;`
Finds the application using an object representing a URL with a `file:` or `eppc:` file scheme; the latter scheme is used for locating remote applications.

`+ (id) applicationWithProcessIdentifier:(pid_t)pid;`
Finds the application using its BSD process identifier (`pid`).

Note: There are initializers corresponding to the above class factory methods—for example, `initWithProcessIdentifier:`—which would require you to allocate a generic `SBApplication` object first. But the recommended usage pattern is to call one of the `applicationWith...` methods.

When you create an `SBApplication` object, Scripting Bridge reads the application's scripting definition and constructs a graph of objects that represents what it finds. It creates an instance representing an application from a dynamically defined and implemented subclass of `SBApplication` that is specific to the application. This instance is the top-level object of the graph. It populates the subordinate objects of the graph with `SBElementArray` and `SBObject` objects. It creates instances of the scripting classes it finds in the application's `sdef` file from dynamically defined and implemented subclasses of `SBObject`. Elements, however, are always represented in Objective-C code by instances of `SBElementArray`, which is a subclass of `NSMutableArray`. This means that you can invoke on `SBElementArray` object all the methods of `NSMutableArray` and its superclass, `NSArray`,

In addition to creating objects, Scripting Bridge implements various methods in the `SBApplication` and `SBObject` subclasses to represent the types of certain items it finds in the application's `sdef` file. It implements scriptable properties as Objective-C declared properties (that is, with the `@property` directive); the declared properties, in turn, synthesize accessor methods to get and (in some cases) set the value of the property. It implements elements as methods that return `SBElementArray` objects. And it implements commands as parameter-less methods returning no value; where these methods are implemented depends on whether they are of a specific or generic object class:

- If it is of a specific object class (such as "document") it is implemented as a method on that class.
- If it is a generic object (such as "specifier") it is implemented as a method of the the `SBApplication` subclass.

Each `SBObject` and `SBApplication` object is built around an object specifier, a reference that tells Scripting Bridge how to locate the actual object in the target application. To obtain the more specific, canonical form of the reference, you must evaluate the object in an appropriate message expression or send it a `get` message. See "Improving the Performance of Scripting Bridge Code" (page 47) for more information on this subject.

Getting Information About an Application's Scripting Definition

You can find out which messages you can send to a scriptable application by examining a header file containing Objective-C declarations of the application class and the application's scripting classes. The header file serves as reference documentation for that application. It includes information about the inheritance relationships between classes and the containment relationships between their objects. It declares commands and elements as methods, and declares properties as Objective-C declared properties. Taking the iTunes application as an

example, the header file shows the definition of the application class (`iTunesApplication`), the application's scripting classes, such as `iTunesTrack` and `iTunesSource`, commands (such as the `eject` method), and properties (such as the `artist` declared property). A header files also includes comments extracted from the scripting definition, such as the comment added to this declaration for the `FinderApplication` class:

```
- (void)empty;   // Empty the trash
```

You need to translate Objective-C method declarations into the Ruby or Python equivalent—or example, replacing the colon of each keyword with an underscore.

To create a header file you need to run two command-line tools—`sdef` and `sdp`—together, with the output from one piped to the other. This is the recommended syntax:

`sdef` *`/path/to/application`*`.app | sdp -fh --basename` *`applicationName`* `--bundleid` *`bundleIdentifier`*

The `sdef` utility gets the scripting definition from the designated application; if that application does not contain an sdef file, but does instead contain scripting information in an older format (such as the scripting suite and terminology property lists), it translates that information into the sdef format first. The `sdp` tool run with the above options generates an Objective-C header file for the designated scriptable application. Thus, for iTunes, you would run the following command to produce a header file named `iTunes.h`:

```
sdef /Applications/iTunes.app | sdp -fh --basename iTunes --bundleid
com.apple.iTunes
```

Improving the Performance of Scripting Bridge Code

Because fetching data from a scriptable application via Apple events is expensive, the Scripting Bridge is designed to defer the sending of Apple events until it needs to. It does this by using references to objects. When you ask the Scripting Bridge for an object in a scriptable application, it returns a reference to that object, not the object itself. It defers evaluation of the reference into its original canonical form until you actually request data from that object. This technique is called lazy evaluation. For example, if you request an iTunes track, it returns a reference to the track object; but when you request the name of the track, it evaluates the reference and sends an Apple event to fetch the string data (that is, the name). This design of the Scripting Bridge leads to a few recommended programming practices:

- Be careful about the order of the statements in your code; do not assume you've received the data that was present in an object when you first obtained a reference to it.
- To force the evaluation of an object reference, invoke the `get` method (declared by `SBObject`) on an object. This call returns the more specific, canonical form of reference to the object.

  Using `get` to force evaluation is valuable when you want to retain a reference to the current object when the non-canonical form of reference—`app.documents[0]` (for frontmost document)—could refer to different objects over time. However, because calling `get` involves the sending of an Apple event, you should use it only when necessary.
- Do not force repeated evaluations of an object reference in a loop, such as when comparing the name of an object against a series of string constants. Each such call results in the sending of an Apple event. Instead force evaluation once and store the returned value in a local variable; then use that variable in the loop.
- As a corollary to the above guideline, avoid enumerating `SBElementArray` objects if there are alternatives, which Scripting Bridge and Cocoa provide:

- Use the `arrayByApplyingSelector:` or `arrayByApplyingSelector:withObject:` method to get a value from each object in the array.
- Use the `makeObjectsPerformSelector:` or `makeObjectsPerformSelector:withObject:` method if you want to make each object in the array do something.
- Use the `filteredArrayUsingPredicate:` method if you want a specific subset of the original array.

Another technique for improving the performance of your code is checking whether an application is launched before trying to communicate with it. When you create an instance of a scriptable application, the Scripting Bridge automatically launches it if it hasn't already been launched. This is an expensive operation. Sometimes this might be what you want, but in other situations you might be interested in communicating with an application only if it's currently being used. In such cases, invoke the `isRunning` method of `SBApplication` and check the returned Boolean value before proceeding.

Scripting Bridge Release Note presents detailed information on lazy evaluation, checking for launched applications, and related APIs and programming guidelines.

Some Examples

To better appreciate the varieties of ways in which you might use Scripting Bridge in RubyCocoa or PyObjC code, let's examine a few examples. The script in Listing 3 creates a proxy instance of the Finder application and from it requests the current contents of the Desktop. Using Ruby regular expressions and string-manipulation methods, it formats and prints these items.

Listing 3 The `Finder_show_desktop.rb` script

```
# Lists the content of the Finder desktop.

require 'osx/cocoa'
include OSX
OSX.require_framework 'ScriptingBridge'

app = SBApplication.applicationWithBundleIdentifier_("com.apple.finder")
ary = app.desktop.entireContents.get
ary.each do |x|
    next unless x.is_a?(OSX::FinderItem)
    puts "#{x.class.name.sub(/^.+::/, '').sub(/_/, ' ').ljust(25)} #{x.name}"
end
```

The script in Listing 4 exchanges data between proxy instances of two applications, TextEdit and Mail. It gets the selected messages in all current Mail viewers and copies each the content of each message to a TextEdit window. There are a couple things of special note in this script. It shows how to create a scripting class for the current application using `classForScriptingClass:` to obtain the `Class` object to use for allocation; it then adds the created document to an `SBElementArray` object (`textedit.documents`) *before* setting its text to that of the email message.

Listing 4 The `get_selected_mail.rb` script

```
# Copy contents of selected Mail messages to a TextEdit window
```

```
require 'osx/cocoa'
include OSX
OSX.require_framework 'ScriptingBridge'

textedit = SBApplication.applicationWithBundleIdentifier_("com.apple.TextEdit")
mailApp = SBApplication.applicationWithBundleIdentifier_("com.apple.mail")
viewers = mailApp.messageViewers
viewers.each do |viewer|
  viewer.selectedMessages.each do |message|
    doc = textedit.classForScriptingClass_("document").alloc.init
    textedit.documents.addObject_(doc)
    doc.setText_(message.content.get)
 end
end
```

Finally. the Listing 5 script updates in the iChat status area the time the system has been running since it was last booted. It is similar to Listing 1 (page 43) it that it makes a system call, but instead of calling the `system` method, it invokes the `uptime` command simply by enclosing it in single quotes. It then formats the output of the command and assigns this formatted string to the iChat `status_message` property. All this occurs in a closed loop, which is re-executed after a five-second pause, which causes a periodic update of the system-uptime message.

Listing 5 The `iChat_uptime.rb` script

```
# Periodically set your iChat status to the output of uptime(1).

require 'osx/cocoa'
include OSX
OSX.require_framework 'ScriptingBridge'

app = SBApplication.applicationWithBundleIdentifier_("com.apple.iChat")
previous_status_message = app.statusMessage
trap('INT') { app.statusMessage = previous_status_message; exit 0 }
while true
    u = `uptime`
    hours = u.scan(/^\s*(\d+:\d+)\s/).to_s + ' hours'
    days = u.scan(/\d+\sdays/).to_s
    app.statusMessage = "OSX up #{days} #{hours}"
    sleep 5
end
```

This script traps interruption of the script (such as happens when the user presses Control-C) and restores the previous value of the iChat status message before exiting.

Generating Framework Metadata

The programmatic interfaces of virtually all frameworks on Mac OS X, even Objective-C frameworks, have ANSI C elements such as functions, string constants, and `enum` constants. The scripting Objective-C bridges, RubyCocoa and PyObjC, can introspect most object-oriented symbols of Objective-C frameworks at runtime, but they cannot introspect ANSI C symbols. Fortunately, there is a utility, `gen_bridge_metadata`, that parses the non-introspectable symbols of framework and libraries and constructs a representation of them that the PyObjC and RubyCocoa bridges can read and internalize at runtime. These generated symbols are defined as XML elements in framework metadata files (also known as BridgeSupport files).

An obvious advantage of framework metadata is that gives the scripting bridges access to the programmatic interfaces of non-Objective-C frameworks, such as Core Foundation, Core Graphics, and Directory Services. Many of the ANSI C frameworks shipped by Apple in Mac OS X v10.5 and later systems include metadata files, and thus their interfaces are accessible from RubyCocoa and PyObjC scripts.

The following sections describe the framework metadata generated to supported the RubyCocoa and PyObjC bridges and explains how to generate metadata files and create the exception files that support that metadata.

The Location and Structure of Framework Metadata Files

Framework metadata is XML markup stored in a file named after the framework and with an extension of `bridgesupport`. Thus, the metadata file for the Application Kit framework (`AppKit.framework`) is named `AppKit.bridgesupport`. Each metadata file describes exactly one framework or dynamically shared library. The RubyCocoa and PyObjC bridges look for metadata files in several places:

- Inside the `Resources` folder of the framework bundle (non-localized) inside a folder named `BridgeSupport`. This is the preferred approach if you own the framework.
- In `/System/Library/BridgeSupport` (this location is reserved for Apple)
- In `/Library/BridgeSupport`
- In `~/Library/BridgeSupport` (that is, in the user's home directory)

The bridges search the locations in the above order and load the first metadata file they find for a given framework. Note that the bridges might also look in the RubyCocoa and PyObjC frameworks for metadata files.

A metadata file consists of several sections of different elements each defining an ANSI C symbol and, in some cases, an Objective-C symbol. The sections include metadata descriptions of string constants, enum constants, functions, structures, opaque objects, classes (with their methods), and informal protocols.

Note: For a detailed description of the content and structure of framework metadata files, see the `BridgeSupport(5)` man page. For an explanation of the Objective-C type-encoding constants used as values in attributes, see "The Runtime System" in *The Objective-C Programming Language*.

At the top of the metadata XML hierarchy is the root element, `signatures`. It has a version attribute.

```
<signatures version='1.0'>
```

Following this is a section dealing with string constants. Listing 1 shows a section of metadata markup that describes string constants by name and encoding type.

Listing 1 Part of the constants section, `AppKit.bridgesupport`

```
<constant name='NSAlternateTitleBinding' type='@'/>
 <constant name='NSAlwaysPresentsApplicationModalAlertsBindingOption' type='@'/>
  <constant name='NSAnimateBinding' type='@'/>
  <constant name='NSAnimationDelayBinding' type='@'/>
  <constant name='NSAnimationProgressMark' type='@'/>
  <constant name='NSAnimationProgressMarkNotification' type='@'/>
  <constant name='NSAnimationTriggerOrderIn' type='@'/>
  <constant name='NSAnimationTriggerOrderOut' type='@'/>
  <constant name='NSAntialiasThresholdChangedNotification' type='@'/>
  <constant name='NSApp' type='@'/>
  <constant name='NSAppKitIgnoredException' type='@'/>
  <constant name='NSAppKitVersionNumber' type='d'/>
  <constant name='NSAppKitVirtualMemoryException' type='@'/>
```

Listing 2shows a section of metadata markup that describes `enum` constants by name and integer value.

Listing 2 Part of the enum section, `AppKit.bridgesupport`

```
  <enum name='NSServiceMalformedServiceDictionaryError' value='66564'/>
  <enum name='NSServiceMiscellaneousError' value='66800'/>
  <enum name='NSServiceRequestTimedOutError' value='66562'/>
  <enum name='NSShadowlessSquareBezelStyle' value='6'/>
  <enum name='NSShiftKeyMask' value='131072'/>
  <enum name='NSShowControlGlyphs' value='1'/>
  <enum name='NSShowInvisibleGlyphs' value='2'/>
  <enum name='NSSingleDateMode' value='0'/>
  <enum name='NSSingleUnderlineStyle' value='1'/>
  <enum name='NSSizeDownFontAction' value='4'/>
  <enum name='NSSizeUpFontAction' value='3'/>
  <enum name='NSSmallCapsFontMask' value='128'/>
  <enum name='NSSmallControlSize' value='1'/>
```

The metadata for functions is more complicated, as it has to describe argument and return types. Listing 3 shows the metadata definition of several functions.

Listing 3 Part of the function section, `AppKit.bridgesupport`

```
<function name='NSDrawBitmap'>
   <arg type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
   <arg type='i'/>
   <arg type='i'/>
   <arg type='i'/>
   <arg type='i'/>
```

```
    <arg type='i'/>
    <arg type='i'/>
    <arg type='B'/>
    <arg type='B'/>
    <arg type='@'/>
    <arg type='^*'/>
  </function>
  <function name='NSDrawButton'>
    <arg type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
    <arg type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
  </function>
  <function name='NSDrawColorTiledRects'>
    <arg type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
    <arg type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
    <arg c_array_length_in_arg='4' type='^i' type_modifier='n'/>
    <arg c_array_length_in_arg='4' type='^@' type_modifier='n'/>
    <arg type='i'/>
    <retval type='{_NSRect={_NSPoint=ff}{_NSSize=ff}}'/>
```

The `gen_bridge_metadata` tool also processes some aspects of Objective-C that the bridges cannot introspect at runtime, such as type modifiers, C-array arguments, and values returned by reference. It also reconciles some aspects that are different in the scripting languages and Objective-C, such as Boolean values. These details mostly derive from exceptions files (see "Creating the Exceptions File" (page 55)). Listing 4 shows a section of metadata specifying methods of the `NSTypesetter` class.

Listing 4 Part of the class and methods section, `AppKit.bridgesupport`

```
  <class name='NSTypesetter'>
    <method selector='usesFontLeading'>
      <retval type='B'/>
    </method>
    <method selector='bidiProcessingEnabled'>
      <retval type='B'/>
    </method>
    <method selector='shouldBreakLineByWordBeforeCharacterAtIndex:'>
      <retval type='B'/>
    </method>
    <method selector='shouldBreakLineByHyphenatingBeforeCharacterAtIndex:'>
      <retval type='B'/>
    </method>
    <method class_method='true'
selector='printingAdjustmentInLayoutManager:forNominallySpacedGlyphRange:packedGlyphs:count:'>
      <arg c_array_length_in_arg='3' index='2' type_modifier='n'/>
    </method>
    <method ignore='true'
selector='getGlyphsInRange:glyphs:characterIndexes:glyphInscriptions:elasticBits:bidiLevels:'/>
    <method ignore='true' selector='substituteGlyphsInRange:withGlyphs:'/>
   <method ignore='true' selector='setLocation:withAdvancements:forStartOfGlyphRange:'/>
```

The last part of a framework metadata file describes any informal protocols in the framework, as illustrated by Listing 5. Again, this information is manually specified in an exceptions file.

Listing 5 Part of the informal protocol section, `AppKit.bridgesupport`

```
 <informal_protocol name='NSDraggingSource'>
    <method type='v20@0:4@8{_NSPoint=ff}12' selector='draggedImage:beganAt:'/>
```

```
    <method type='v24@0:4@8{_NSPoint=ff}12c20'
selector='draggedImage:endedAt:deposited:'/>
    <method type='v24@0:4@8{_NSPoint=ff}12I20'
selector='draggedImage:endedAt:operation:'/>
    <method type='v20@0:4@8{_NSPoint=ff}12' selector='draggedImage:movedTo:'/>
    <method type='I12@0:4c8' selector='draggingSourceOperationMaskForLocal:'/>
    <method type='c8@0:4' selector='ignoreModifierKeysWhileDragging'/>
    <method type='@12@0:4@8'
selector='namesOfPromisedFilesDroppedAtDestination:'/>
  </informal_protocol>
  <informal_protocol name='NSDrawerDelegate'>
    <method type='c12@0:4@8' selector='drawerShouldClose:'/>
    <method type='c12@0:4@8' selector='drawerShouldOpen:'/>
    <method type='{_NSSize=ff}20@0:4@8{_NSSize=ff}12'
selector='drawerWillResizeContents:toSize:'/>
  </informal_protocol>
```

Using the gen_bridge_metadata Tool

The `gen_bridge_metadata` tool parses framework header files and runs the `gcc` compiler on a framework binary to extract the public symbols. With this data, it composes an XML metadata file for the specified framework. The simplest form of the command requires only the name of the framework (minus the `framework` extension) and the name of the output file:

```
$> gen_bridge_metadata -f MyFramework -o MyFramework.bridgesupport
```

For this shorthand reference to a framework to work, the framework must be installed in one of the standard file-system locations: `/System/Library/Frameworks`, `/Library/Frameworks`, `/Network/Library/Frameworks`, or `~/Library/Frameworks`. If the framework is located elsewhere, you can specify an absolute path to the framework instead.

Most frameworks require a manually prepared exceptions file to complete the framework metadata. You specify this file on the command line with the `-e` option:

```
$> gen_bridge_metadata -f MyFramework -e MyFrameworkExceptions.xml -o
MyFramework.bridgesupport
```

For more about the exceptions file, see "Creating the Exceptions File" (page 55).

Framework metadata files cannot describe inline functions in a form that the bridges can use. If your framework has inline functions, you therefore also need to generate a dynamically shared library, which the bridges can use. The file extension for the created file should be `dylib`. The following is an example command:

```
$> gen_bridge_metadata -f MyFramework -F dylib -o MyFramework.dylib
```

The `-F` option is for specifying a format, one of "final", "exceptions-format", or "dylib". The default format is "final".

For more information on `gen_bridge_metadata` consult the `gen_bridge_metadata(1)` man page. You can also run the tool with an argument of `-h` (or `--help`) to get a list of options.

Creating the Exceptions File

You might have to supplement the metadata for your framework with an exceptions file. An exceptions file records aspects of a framework's programmatic interface that the bridges cannot introspect at runtime or that conflict with something in a scripting language. These items include type modifiers, C-array arguments, informal protocols, values returned by reference, and Boolean values.

First, you need to create an exception template, which will provide the structure of the XML file. Run the following at the command line to create the exceptions template:

```
gen_bridge_metadata -f MyFramework -F exceptions-template -o
MyFrameworkExceptions.xml
```

Next open the template file in a text editor and insert your framework-specific information in the appropriate places.

Note: Instructions on completing an exceptions file will be provided in a future version of this document. For now, you can consult the `BridgeSupport(5)` manual page.

When your exception file is complete, you can generate the final bridge support file for your framework, as described in "Using the gen_bridge_metadata Tool" (page 54). Make sure that you supply the `-e` parameter and the path to the exceptions file. The `gen_bridge_metadata` tool will fail if your exception file contains any errors.

Creating Your Own Bridge

Beginning in Mac OS X version 10.5, you can easily create your own bridge between Objective-C and any language. You can use the generated bridge support files and the libffi library to have your bridge call C functions and create C closures in a dynamic, architecture-agnostic way. Libffi provides a bridge from interpreted code to compiled code that can tell the interpreter at runtime the number and types of function arguments and return values.

You can learn more about libffi by reading the manual pages for `ffi`, `ffi_prep_cif`, `ffi_prep_closure`, and `ffi_call`.

Document Revision History

This table describes the changes to *Ruby and Python Programming Topics for Mac OS X*.

Date	Notes
2007-10-31	New document that describes Ruby and Python on Mac OS X, and especially the bridges between them and Objective-C and Open Scripting Architecture.