

Extracted from:

Programming Cocoa with Ruby

Create Compelling Mac Apps Using RubyCocoa

This PDF file contains pages extracted from Programming Cocoa with Ruby, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2009 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The
Pragmatic
Programmers

Programming Cocoa with Ruby

Create Compelling Mac Apps
Using RubyCocoa



Brian Marick

Edited by Daniel H Steinberg

The Facets  of Ruby Series



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

<http://www.pragprog.com>

Copyright © 2009 Brian Marick.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-19-0

ISBN-13: 978-1-934356-19-7

Printed on acid-free paper.

P1.0 printing, July 2009

Version: 2009-8-7

Try This Yourself

You can add text to the status bar with `setTitle`. Try that in `statusbar-item.rb`, both with and without an accompanying image.³

2.3 Menus

Our status bar item doesn't do anything, so let's give it a menu. For fun, I'll use it to make the app speak to us. That's not hard: I'll use a Cocoa object, `NSSpeechSynthesizer`, to turn text into speech.

Before starting that, let's separate concerns. App will concern itself only with application-wide events such as being launched and being terminated. A new class, `SpeechController`, will do everything else.

Here's the new version of App:

[Download](#) `statusbar/speaking-statusbar.rb`

```
class App < NSObject
  def applicationDidFinishLaunching(aNotification)
    statusbar = NSSStatusBar.systemStatusBar
    status_item = statusbar.statusItemWithLength(NSVariableStatusItemLength)
    image = NSImage.alloc.initWithContentsOfFile("stretch.tiff")
    status_item.setImage(image)
    ❶ SpeechController.alloc.init.add_menu_to(status_item)
  end
end
```

Only one thing has changed, at line ❶. We just create a `SpeechController`, ask it to add its menu to the status bar item, and then forget about it. Notice that a `SpeechController` is an Objective-C object—you can tell because it's created with `alloc` and `init`.

And here's the `SpeechController` class:

[Download](#) `statusbar/speaking-statusbar.rb`

```
class SpeechController < NSObject
  def init
    ❶ super_init
    @synthesizer = NSSpeechSynthesizer.alloc.init
    ❷ self
  end
end
```

Like App, `SpeechController` descends from `NSObject`. A `SpeechController` needs to define its own `init`, though, because we want it to create an

3. If you're not working in the `statusbar` directory, get a copy of `statusbar/stretch.tiff` from there before running the script.

NSSpeechSynthesizer and hold onto it in an instance variable. Such an `init` method differs from Ruby’s familiar `initialize` in two ways:

- ❶ In an ordinary Ruby class, the `initialize` method uses `super` to call its superclass’s `initialize` method. In an NSObject subclass, `init` calls the superclass’s `init` method with `super_init`. (In general, any overriding method *method* calls its superclass version with `super_method`.)

As you saw on page 28, `init` methods can sometimes return `nil`. For that reason, a pedantically safe use of the superclass would look like this:

```
return nil unless super_init
```

In this case, though, I know that NSObject’s `init` always returns `self`. (In fact, it does nothing *but* return `self`, so I could omit the line entirely.)

- ❷ In an ordinary Ruby class, `initialize`’s return value is irrelevant. In contrast, an NSObject subclass *must* return `self` (or, in the case of error, `nil`). If I’d forgotten line ❷, code like this:

```
s = SpeechController.alloc.init
s.add_menu_to(status_item)
```

... would make `s` an NSSpeechSynthesizer and then blow up on the next line with a “no such message” failure. Even after seeing a lot of those failures, it still sometimes takes me much too long to think of blaming `init`.

Now for the menu. In Cocoa, a menu is represented by an NSMenu that contains NSMenuItem objects. It’s those objects that receive “you’ve been clicked” events from the window manager. If an NSMenuItem handles the event, it forwards the work by calling an *action method* attached to a *target object*. (See Figure 2.2, on the following page.)

The NSMenu itself does only a little work. It asks each item for its name and *key equivalent* (the keystroke that selects that item via the keyboard instead of the mouse). Then it paints all the items on the screen.

SpeechController’s `add_menu_to`, shown in Figure 2.3, on the next page, wires all this together.

It begins (❶) by allocating an NSMenu object and attaching it to whatever container was given. This is another example of duck typing (and a benefit of separation of concerns): this particular class doesn’t care

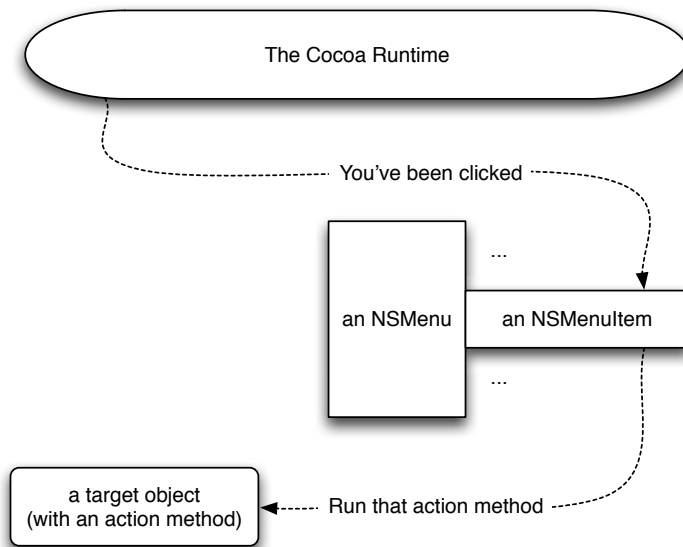


Figure 2.2: Clicking a menu

[Download](#) `statusbar/speaking-statusbar.rb`

```

def add_menu_to(container)
  ❶ menu = NSMenu.alloc.init
    container.setMenu(menu)

  ❷ menu_item = menu.addItemWithTitle_action_keyEquivalent(
    "Speak", "speak:", '')
  ❸ menu_item.setTarget(self)

  menu_item = menu.addItemWithTitle_action_keyEquivalent(
  ❹ "Quit", "terminate:", 'q')
  ❺ menu_item.setKeyEquivalentModifierMask(NSCommandKeyMask)
  ❻ menu_item.setTarget(NSApp)
end

  ❼ def speak(sender)
    @synthesizer.startSpeakingString("I have nothing to say.")
  end
end

```

`statusbar/speaking-statusbar.rb`

Figure 2.3: Building a menu

what it's attached to, so long as that object responds to `setMenu`. Today, it's a status bar item. Tomorrow, it could be something else.

Next, an `NSMenuItem` is created and assigned to the menu by `addItemWithTitle_action_keyEquivalent` (line ②). What's up with *that* name? Objective-C has an interesting and nearly unique way of naming methods. Here's (almost) what Objective-C code that added a menu item would look like:⁴

```
[menu addItemWithTitle: "Speak" action: "speak:" keyEquivalent: ""]
```

The method being called here is named `addItemWithTitle:action:keyEquivalent:`. It takes exactly three arguments that have to come in exactly the defined order.

RubyCocoa has to provide you with a way of naming that Objective-C method. It can't use the same name, because method names in Ruby can't contain colons. So, the colons are replaced with underscores. To avoid excessive ugliness, you can leave off the last underscore, as I did at line ②.⁵

The first and third arguments to the method provide the name and key equivalent. (This particular item has no key equivalent.) The second argument is the name of the message to send when the menu item is selected. Although the `speak` method is defined in Ruby, I've used Objective-C's notion of its name: `"speak:"`. The name ends in a colon because (as you'll see shortly), `speak` takes a single argument.

Which object receives the `speak:` message is set on the next line (③). In this case, the `SpeechController` handles the message itself.

Lines ④ and ⑤ show how you create a keyboard equivalent. Those are almost never plain characters like `q`. They're usually characters with modifiers, like `Command-Q`. For whatever reason, the character and modifier keystrokes are set in separate methods.

The menu item will send a `terminate: message`, but not to `SpeechController`. Since it's a message about the whole app, it's targeted at `NSApp` (line ⑥), an Objective-C class that implements `terminate:`.

4. I've removed a little type casting because it's not important to this explanation. To be pedantic, the title and key equivalent shouldn't be strings. They should be `NSString` objects, which are written as `@string`. Similarly, the action argument should be a "selector," not a string. You'll see more—and more correct—examples of Objective-C later in the book.

5. That's not always safe: consider an Objective-C class that has two methods, `action` and `action:`.

The speak (🗨️) action is simple. Notice that it takes a sender argument, which will be the NSMenuItem that was clicked. Action methods can use the sender to query or change the user interface.

If you run the app, you'll probably notice that the synthesizer takes a second or two to start talking after you click the menu item. Presumably it's doing some first-time initialization. It's more prompt the second time.

Try This Yourself

1. Put this at the end of speak:

```
puts sender.objc_methods.grep(/title/i)
```

Use one or more of those methods to change the menu after something is said.

2. While terminating, NSApp will send its delegate two messages: `applicationShouldTerminate` and `applicationWillTerminate`. The first lets the delegate decide to cancel shutdown, and the second gives it a chance to do any of its own cleanup.

Use `applicationWillTerminate` to print out “Goodbye, cruel world!”

3. Make `applicationShouldTerminate` return `false` unless the app has spoken at least twice, `true` otherwise. See what happens when you return values like `nil`, `“fred”`, and the integer `0`.

A small quirk: unlike the delegate messages you've seen so far, `applicationShouldTerminate` takes an `NSApplication` as its argument, so `sender` or `app` would be a better name than `aNotification`.

(If you need help, there's a solution in `statusbar/speaking-statusbar-solution.rb`.)

2.4 An Application Bundle

Fine though our script may be, it doesn't behave like a Mac application. If you double-click it, it doesn't launch. (Most likely, it opens in an editor.) It doesn't get an icon in the Dock, you can't see it if you `⌘-Tab` through open applications, and so on. In this section, I'll explain what's special about apps. You'll create your first one in Chapter 3, *Working with Interface Builder and Xcode*, on page 39.

The Pragmatic Bookshelf

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

Visit Us Online

Programming Cocoa with Ruby's Home Page

<http://pragprog.com/titles/bmrc>

Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates

<http://pragprog.com/updates>

Be notified when updates and new books become available.

Join the Community

<http://pragprog.com/community>

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

<http://pragprog.com/news>

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

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

Contact Us

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