

Extracted from:

iOS SDK Development

This PDF file contains pages extracted from *iOS SDK Development*, 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 © 2010 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 Bookshelf

Dallas, Texas • Raleigh, North Carolina

iOS SDK Development



Chris Adamson
and Bill Dudney
Edited by Brian P. Hogan

iOS SDK Development

Chris Adamson
Bill Dudney

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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, PragProg 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://pragprog.com>.

Copyright © 2011 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.

Printed in the United States of America.

ISBN-13: 978-1-934356-94-4

Printed on acid-free paper.

Book version: B1.0—December 14, 2011

3.1 Blocks

As it stands right now, our Twitter app lets us compose a tweet, view our Twitter page, and operate in multiple languages. But when we compose a Tweet and click “Send”... nothing happens. We have to manually click the “Show my Tweets” button to see if the Tweet went out. Surely we can improve that behavior.

If we take a look at the `TWTweetComposeViewController`, we find a property called `completionHandler`, described as “the handler to call when the user is done composing the tweet”. So that’s good: we could use this to reload the web view once the Tweet has been sent. Notice that like the event handlers for the buttons, this is an asynchronous concern: we are providing code that gets called only when an unpredictable user-interface event demands it.

Let’s get ready by doing a little refactoring. We will want to reload the web view either in response to a user tapping the reload button, or when he or she finishes composing a Tweet. That calls for moving the reload logic into its own method, `reloadTweets`, which can then be called from `handleShowMyTweetsTapped`: and from our completion handler.

To introduce a new method we can call in multiple places, we need to declare the method signature prior to any calls that actually use it. We could declare it in the header file, but that exposes it publicly, and it’s really more of an implementation detail. What we need is a C-style forward declaration in the `.m` file, but Objective-C doesn’t support them. What we use instead is a *class extension*, which looks like a second `@interface` declaration, but goes inside the implementation file. In `PRPViewController.m`, before the `@implementation`, add the following:

```
Download Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m
@interface PRPViewController()
-(void) reloadTweets;
@end
```

This extends the previous declaration of the `PRPViewController` interface, the one in the `.h` file, by adding another method declaration, `reloadTweets`. This method can be called anywhere in the `.m` file, but isn’t exposed to outsiders. Now go to the bottom of the file and refactor the event-handler to call `reloadTweets`:

```
Download Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m
-(IBAction) handleShowMyTweetsTapped: (id) sender {
    [self reloadTweets];
}

-(void) reloadTweets {
```

```

[self.twitterWebView loadRequest:
 [NSURLRequest requestWithURL:
 [NSURL URLWithString:@"http://www.twitter.com/pragprog"]]];
}

```

Now we just need to call `reloadTweets` when the user sends the Tweet, which we get from this completion handler. The docs tell us that the handler is of type `TWTweetComposeViewControllerCompletionHandler`, so we follow the link to that typedef and it looks like this:

```

typedef void (^TWTweetComposeViewControllerCompletionHandler)
(TWTweetComposeViewControllerResult result);

```

What... the... heck?

The carat character tells us we're not in Kansas anymore. This indicates a new C-language extension introduced by Apple, called a *block*. A block is an object that contains both executable code and program state. The idea is much like that of a *closure*: the code inside the block receives a copy of the variables that were in scope when the block was created.

For this typedef, the `void` tells us that the block does not return a value, and it accepts a `TWTweetComposeViewControllerResult` called `result` as a parameter. The result is an enum with values that tell us whether the `TWTweetComposeViewController` was dismissed with the “Cancel” or the “Done” button.

So, to reload the web view, we need to create a block that checks to see that the result for is `TWTweetComposeViewControllerResultDone` and if so, calls `reloadTweets`. We create a block with the carat (^) character, like the documentation did. Rewrite `handleTweetButtonTapped`: as follows:

Download `Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m`

```

Line 1 -(IBAction) handleTweetButtonTapped: (id) sender {
-   if ([TWTweetComposeViewController canSendTweet]) {
-       TWTweetComposeViewController *tweetVC = [[TWTweetComposeViewController alloc]
-                                               initWith];
5       [tweetVC setInitialText: NSLocalizedString (
-           @"I just finished the first project in iOS SDK Development. #pragsios",
-           nil)];
-       tweetVC.completionHandler = ^(TWTweetComposeViewControllerResult result) {
-           if (result == TWTweetComposeViewControllerResultDone) {
10          [self reloadTweets];
-           }
-       };
-       [self presentViewController:tweetVC animated:YES];
-   }
15 }

```

Categories

Class extensions are actually just a special case of *categories*, which allow us to add methods to any class, even classes we don't own, like those in the iOS SDK frameworks.

For example, consider that `NSArray` has a `lastObject` method, but doesn't have one to get the first object. `[myArray objectAtIndex:0]` isn't equivalent, because it throws an exception for empty arrays. With a category, we could write a safe `firstObject` method, by declaring a category of new methods on `NSString` like this:

```
@interface NSString (MySafeMethods)
-(id) firstObject;
@end
```

By convention, this header would go in a file called `NSString+MySafeMethods.h`, and the implementation would go in a corresponding `.m`. The implementation can't add instance variables, so there are limits on what we can do in a category.

A class extension, like we use in this example, is a just category without a name (indicated by empty parentheses) that lives in a `.m` file and therefore isn't public. We're going to use them a lot, to keep our private methods private and keep our headers uncluttered.

The new part is lines 8 through 12. We create a block with the `^` and the parameter list from the documentation, and then enclose our code in curly braces. On line 9, we test the value of the result parameter. If it's `TWTweetComposeViewControllerResultDone`, then we call `[self reloadTweets]` on line 10. The interesting part of this line is where we get `self` from, since it isn't passed into the block as a parameter. It's a variable in scope at the time of the block's creation, so the code in our block can call it directly. For that matter, our block could also refer to the sender parameter that was sent as a parameter to `handleTweetButtonTapped:`, because that's another variable that's in scope when the block is created.

Run this and send a Tweet. Shortly after "Send" is tapped, the block gets executed and its code runs, automatically reloading the web view without any further action by the user.

The iOS SDK uses blocks in several interesting ways. The pattern seen here, the *completion handler*, is a clean way of determining what should happen when a long-running action like network access or media I/O completes. Foundation's collection classes also make substantial use of blocks. `NSArray` has a method `enumerateObjectsUsingBlock:` that runs a block against every member of an array, and `NSDictionary` has similar methods that run a block on every

key-value pair. We can also use a block as the sorting criteria for the contents of an NSArray and the keys of an NSDictionary.