Extracted from:

iOS SDK Development

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS SDK Development

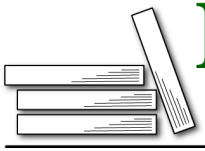
Chris Adamson
and Bill Dudney

Edited by Brian P. Hogan

iOS SDK Development

Chris Adamson
Bill Dudney

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-94-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2012

3.1 Encapsulating Concurrent Code with 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 SLComposeViewController in the documentation, we find a property called completionHandler, described as "the handler to call when the user is done composing a post." 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.

Prior to Xcode 4.4, we needed to declare the method signature prior to any calls that actually use it, and this is still a useful practice for backward compatibility. We could declare the method 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, an empty class extension has been stubbed out for us; edit it to look like this:

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 it isn't exposed to outsiders.

Speaking of exposure to other classes, there's no good reason that the twitterWebView should be publicly visible, and we certainly wouldn't want a caller to be able to reassign it to an object other than the one we created in our nib.

Fortunately, class extensions can also contain property declarations. So cut the @property line from the PRPViewController.h header file, and paste it into the .m file's class extension, which should now look like this:

Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m

```
@interface PRPViewController()
-(void) reloadTweets;
@property (nonatomic, strong) IBOutlet UIWebView *twitterWebView;
@end
```

Now that we've hidden the property, let's get back to our original refactoring. Go to the bottom of the file and refactor the event handler to call reloadTweets:

Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m

```
-(IBAction) handleShowMyTweetsTapped: (id) sender {
  [self reloadTweets];
}

-(void) reloadTweets {
  [self.twitterWebView loadRequest:
   [NSURLRequest requestWithURL:
    [NSURL URLWithString:@"http://www.twitter.com/yourhandle"]]];
}
```

Categories

Class extensions can be seen as a special case of categories, one of Objective-C's most clever features. Categories 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 it 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 NSArray like this:

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

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

Unlike a category, a class extension *can* provide instance variables in that it supports property declarations. In fact, now that the Xcode compiler can find method calls without the forward declaration that a class extension provides, the private declaration of properties is probably the most significant use of class extensions.

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

```
typedef void (^SLComposeViewControllerCompletionHandler)
        (SLComposeViewControllerResult 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 SLComposeViewControllerResult called result as a parameter. The result is an enum with values that tells us whether the SLComposeViewController 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 is SLComposeViewControllerResultDone, and if so, calls reloadTweets. We create a block with the carat (^) character, like the documentation did. We rewrite handleTweetButtonTapped: as follows:

Concurrency/PRPFirstProjectTweeter04/PRPFirstProjectTweeter/PRPViewController.m

```
Line 1 -(IBAction) handleTweetButtonTapped: (id) sender {
   -   if ([SLComposeViewController isAvailableForServiceType: SLServiceTypeTwitter]) {
   -     SLComposeViewController *tweetVC =
   -     [SLComposeViewController composeViewControllerForServiceType:
   5      SLServiceTypeTwitter];
   -     [tweetVC setInitialText: NSLocalizedString (
   -       @"I just finished the first project in iOS SDK Development. #pragsios",
   -       nil)];
   -     tweetVC.completionHandler = ^(SLComposeViewControllerResult result) {
  10       if (result == SLComposeViewControllerResultDone) {
   -         [self dismissViewControllerAnimated:YES completion:NULL];
   -         [self reloadTweets];
   -       }
   -     };
  15     [self presentViewController:tweetVC animated:YES completion:NULL];
   -   }
   - }
```

The new part is lines 9 through 14. We create a block with the ^ and the parameter list from the documentation and then enclose our code in curly braces. On line 10, we test the value of the result parameter. If it's SLComposeViewControllerResultDone, then we know the tweet was sent (and not cancelled) and we get to work. We start with a call to [self dismissViewControllerAnimated: YES completion:NULL] on line 11, which takes a view controller to dismiss (self) and a completion block to execute when the dismissal animation completes (the empty block NULL means we do nothing special).

We didn't have to explicitly dismiss the SLComposeViewController before we added the custom completionHandler, but now that we're using a SLComposeViewControllerCompletionHandler, its documentation tells us that "the completion handler is called while the SLComposeViewController is still visible and *it is responsible for dismissing the view controller*" (emphasis ours).

To fill in the web view, we call [self reloadTweets] on line 12. 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.

Notice the odd syntax on line 14 (};). The closing curly brace ends the block that started on line 9, and the semicolon ends the assignment of this block to the property tweetVC.completionHandler. This may be harder to read than it is to write; syntax-aware code completion in Xcode helps a lot with blocks.

Run this and send a tweet. Shortly after Send is tapped, the Social framework sends the tweet and then executes the block, which automatically reloads 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. We actually had an option to use blocks way back in the first version of this project: the third argument of presentViewController:animated:completion: specifies a completion handler block to execute once a modal view controller's view has been shown. We've used NULL because we don't need to do anything special once the tweet composer appears. 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.