Extracted from:

iPhone SDK Development

Building iPhone Applications


The Pragmatic Programmers
Add Book
Save
iPhone SDK Development
Bill Dudney and
Chris Adamson
Edited by Daniel H Steinberg
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
space
return
.?123

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

Printed in the United States of America.

ISBN-10: 1-934356-25-5

ISBN-13: 978-1-934356-25-8

Printed on acid-free paper.

P2.0 printing, December 2009

Version: 2010-2-3

Here's the default implementation in tableView:cellForRowAtIndexPath::[3]

Download TableViews/MovieTable01/Classes/RootViewController.m

```
Line 1  static NSString *CellIdentifier = @"Cell";
    2  UITableViewCell *cell =
    3          [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    4  if (cell == nil) {
    5  cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
    6                  reuseIdentifier:CellIdentifier] autorelease];
    7  }
```

Line 1 creates a *cell identifier*, a string that indicates the kind of cell we want. The idea here is that if you use different styles of cells in the same table (either default styles or layouts of your own creation), you will need to distinguish them in the table's cache so you get back the style of cell you need. In the default case, you use only one style, so any arbitrary string like "Cell" will suffice. Next, lines 2–3 attempt to *dequeue* a cell, that is to say, to retrieve a cell from the table's cache, passing in the identifier to indicate what kind of cell is needed. If this fails, then a new cell is allocated and initialized.

5.5 Editing Tables

So now, we've covered how to provide table contents and gain some control over how the contents of a cell are presented. The next step is to make the table editable. What this really means is that we want to make the table serve as an interface for editing the underlying model. When we delete a row in the table, we want to delete the object from the model, and when we add an item to the model, we want the table updated to reflect that.

Let's start with deletes, which are easier. In fact, the commented-out code provided by the navigation-application template includes the basics of what we need to provide deletion. Start with tableView:canEditRowAtIndexPath:. The default implementation (and the default behavior, if this UITableViewDataSource method is not implemented at all) is to not permit editing of any row. Uncomment the default implementation, and change it to return YES;.

3. We've reformatted the default code to fit the layout of this book.

To implement the delete, we need to implement tableView:commitEditingStyle:forRowAtIndexPath:. The commented-out implementation has an **if-then** block for handling cases where the editing style is UITableViewCellEditingStyleDelete and UITableViewCellEditingStyleInsert. We need to support the latter only. To perform a delete, we need to do two things: remove the indicated object from the moviesArray model, and then refresh the on-screen UITableView. For the former, UITableView provides the method deleteRowsAtIndexPaths:withRowAnimation:, which is exactly what we need. Add the highlighted line to the default implementation, as shown here, and delete the **else** block for UITableViewCellEditingStyleInsert:

Download TableViews/MovieTable01/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
               commitEditingStyle: (UITableViewCellEditingStyle)editingStyle
               forRowAtIndexPath:(NSIndexPath *)indexPath {
        if (editingStyle == UITableViewCellEditingStyleDelete) {
               // Delete the row from the data source.
               [moviesArray removeObjectAtIndex: indexPath.row];
               [tableView deleteRowsAtIndexPaths:
                                [NSArray arrayWithObject:indexPath]
                      withRowAnimation:UITableViewRowAnimationFade];
        }
}
```

This gives us swipe-to-delete behavior, but some users don't even know it exists. Fortunately, since we're a navigation app, we have a navigation bar at the top of the screen that is well suited to hosting an Edit button. As in other apps, its default behavior when active is to add an "unlock to delete" button to the left side of every table row that allows editing, which brings up the right-side Delete button when tapped.

In the viewDidLoad method you uncommented, you might have noticed the following commented-out code:

Download TableViews/MovieTable01/Classes/RootViewController.m

```
    // Uncomment the following line to display an Edit button in the
    // navigation bar for this view controller.
// self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

You might recall from Section 5.2, *Setting Up Table-Based Navigation*, on page 90 that in MainView.xib, the RootViewController came set up with UINavigationItem as a child element. That represents the blue bar above the table, typically used for forward-back navigation and for editing tables. It has two properties for setting buttons in the bar: leftBarButtonItem and rightBarButtonItem. Then, on the right side of this assignment, notice the reference to self.editButtonItem. Every UIViewController

supports this editButtonItem property, which returns a UIBarButtonItem that calls the view controller's setEditing:animated: method and toggles its state between Edit and Done.

The commented-out line is almost what we want, but let's put the Edit button on the left, so we can leave the right side for an Add button that we'll create later. So, here's the line you'll need in viewDidLoad:

Download TableViews/MovieTable01/Classes/RootViewController.m

```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

Once you Build and Go, you should now be able to tap the Edit button and bring up the unlock-to-delete button for all the rows. In Figure 5.5, on the next page, we can see the table in editing mode (with some more sample data to fill out its rows).

5.6 Navigating with Tables

Our next task is to allow the user to add a table row. In the previous chapter, we developed a MovieEditorViewController, and that's perfectly well suited to entering the fields of a new Movie object or editing an existing one. And once created, it would be simple enough to add the new Movie object to the model and update the table.

So, where do we put the editor? In the previous chapter, we used the UIViewController method presentModalViewController:animated: to slide in the editor. In this case, we're going to learn something new: how to use the navigation objects at our disposal. We created the project as a navigation-based application in part because it gave us a good starting point for our table, and navigation also turns out to be a good idiom for switching between our viewing and editing tasks.

Navigation on the iPhone uses a "drill-down" metaphor that you are probably familiar with from the Mail, iPod/Music, and Settings applications. In the SDK, this is managed by a UINavigationController, which maintains the navigation state as a stack of view controllers. Every time you drill down, you push a new UIViewController onto the stack. When you go back, you pop the current view controller off the stack, returning to the previous view. The navigation is handled in code, independent of how it's represented on-screen: whether you navigate by tapping rows in a table or buttons in the navigation bar, the underlying stack management is the same.

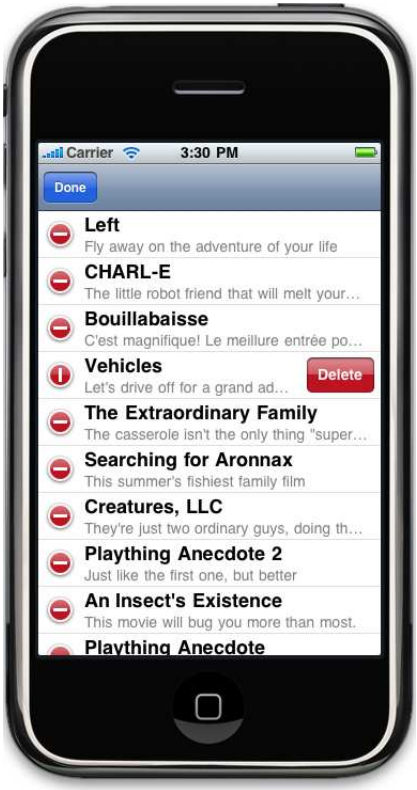

Figure 5.5: Using the default editButtonItem to delete rows from a UITableView

Adding the MovieEditorViewController

To try this, let's get to the MovieEditorViewController by means of the navigation API. In fact, we'll use it for two purposes: to edit items already in the table and to create new items.

As with the Movie class, you'll need to copy the MovieEditorViewController.h and MovieEditorViewController.m files to your project's Classes folder and then add those copies to the Xcode project. Also copy over the MovieEditorViewController.xib (with Add > Existing Files as before) to the project's Resources group. In the earlier examples, this editor view was presented modally and took up the whole screen. In this application, it's part of the navigation, and therefore the navigation bar will take up some space above the view. Fortunately, Interface Builder lets us simulate a navigation bar to make sure everything still fits in the view. Open

the nib in IB, select the view, and bring up its Property inspector (⌘ 1). Under Simulated Interface Elements, set Top Bar to Navigation Bar to see how the view will look as part of the navigation. In this case, the Done button won't be pushed off-screen, but you might want to adjust its position to get it inside IB's dashed margin.

To bring up the movie editor, our RootViewController needs to push an instance of the MovieEditorViewController on to the navigation stack. We could create the view controller in code, but since we only ever need one instance, it makes sense to create it in Interface Builder. The first step, then, is to create an IBOutlet in RootViewController.h. Add an instance variable MovieEditorViewController* movieEditor; inside the **@interface**'s curly-brace block, and then declare the property as an outlet after the close brace:

Download TableViews/MovieTable01/Classes/RootViewController.h

```
@property (nonatomic, retain) IBOutlet MovieEditorViewController *movieEditor;
```

As usual, you'll need to **@synthesize** this property in the .m file. Also, remember to put #import "MovieEditorViewController.h" in the header.

Now you're ready to create an instance of MovieEditorViewController in Interface Builder. Open RootViewController.xib with IB, and drag a UIViewController from the Library into the nib document window. Select this view controller, and use the Identity inspector (⌘ 4) to set its class to MovieEditorViewController. The last step is to connect this object to the outlet you just created. Ctrl+click or right-click File's Owner (or show its Connections inspector with ⌘ 2), and drag a connection from movieEditor to the view controller object you just created. We're done with IB for now, so save the file.

Editing an Existing Table Item

Let's start by using the MovieEditorViewController to edit an item in the table. When the user selects a row, we'll navigate to the editor and load the current state of the selected Movie object into the editor.

The first thing we need to do is to react to the selection event. The UITableViewDelegate gets this event in the delegate method tableView:didSelectRowAtIndexPath:. The navigation-application template provides a commented-out version of this method in RootViewController, though its sample code creates a new view controller programatically. We don't need to do that, since we already have the next view controller. It's the movieEditor that we just set up in Interface Builder. So, we just need to set up that view controller and navigate to it.

Declare an instance variable of type Movie* named editingMovie in the header file. It remembers which Movie object is being edited, so we'll know what to update in the table when we navigate to the table. Once you've done that, the steps here are pretty simple. Remember what movie we're editing, tell the MovieEditorViewController what movie it's editing, and navigate to that view controller with the UINavigationController's pushViewController:animated: method.

Download TableViews/MovieTable01/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
                  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
        editingMovie = [moviesArray objectAtIndex:indexPath.row];
        movieEditor.movie = editingMovie;
        [self.navigationController pushViewController:movieEditor animated:YES];
}
```

What's interesting about the last step is how we get a reference to the navigation controller. . . remember, we haven't defined an ivar or property for it; in fact, the navigation controller was created for us in MainWindow.xib, and we haven't touched it with IB. The neat trick is the navigationController property defined by the UIViewController class and therefore inherited by RootViewController. This property (also callable as an instance method) looks through the object hierarchy to find a parent or ancestor object that is a UINavigationController. Thanks to this method, you never need to explicitly make connections to your navigation controller. Your root view controller and any view controllers it pushes onto the navigation stack can get to the navigation controller with this property, using it to navigate forward or back or to update the on-screen navigation bar.

This is all we need to do to the movie editor view; now we need a way to get back from the editor to the root. MovieEditorViewController has a done method that's connected in IB to the Done button,[4] but its implementation needs to be updated. Instead of dismissing itself as a modal view controller, it needs to navigate back to the previous view controller:

Download TableViews/MovieTable01/Classes/MovieEditorViewController.m

```
- (IBAction)done {
        [self.navigationController popViewControllerAnimated:YES];
}
```

4. If we didn't already have a Done button in the view, it would be more typical to set up a Done or Back button in the navigation bar. The navigation in the example in Chapter 8, *File I/O*, on page 140 will work like this.

As you can see, the MovieEditorViewController also can use the inherited navigationController property to get the UINavigationController.

This will navigate to and from the movie editor; the only task left to attend to is to update the table when we return from an edit. The RootViewController will get the viewWillAppear: callback when we navigate back to it, so we can use that as a signal to update the table view:

Download TableViews/MovieTable01/Classes/RootViewController.m

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
        // update table view if a movie was edited
        if (editingMovie) {
                NSIndexPath *updatedPath = [NSIndexPath
                        indexPathForRow: [moviesArray indexOfObject: editingMovie]
                        inSection: 0];
                NSArray *updatedPaths = [NSArray arrayWithObject:updatedPath];
                [self.tableView reloadRowsAtIndexPaths:updatedPaths
                                withRowAnimation:NO];
                editingMovie = nil;
        }
}
```

We gate our update logic with a check to see whether a movie is being edited, since this method will also be called at other times (at startup, for example). If we are returning from an edit, we need to identify the one table row being updated. We can figure this out by getting the array index that corresponds to editingMovie, constructing an NSIndexPath that goes to that row in section 0 of the table, and pass the path to the table view's reloadRowsAtIndexPaths:withAnimation: method.

Adding an Item to the Table

Another thing we'd like to support is the ability to add new items to the table. We can actually make this a special case of editing. When the user taps an Add button, we quietly add an empty Movie to the table model, insert a table row, and navigate to the editor.

Previously, we used the navigation bar's leftBarButtonItem for the provided editButtonItem, so let's put the Add button on the right side of the navigation bar. We don't inherit an Add button from UIViewController like we did with the Edit button, so we'll create one ourselves.

First, go to RootViewController.h, and set up an IBAction to handle an event from the button we're about to create:

Download TableViews/MovieTable01/Classes/RootViewController.h

```
-(IBAction) handleAddTapped;
```

Now, since we need to work with the navigation objects that Xcode created for us, we'll use Interface Builder to open the MainWindow.xib file, where they're defined. Switch the view mode in the nib document window to list or column view, and double-click the Navigation Controller object. This will bring up a window with the navigation bar at the top and a view placeholder at the bottom that says it's loaded from RootViewController. You'll notice that the Edit button is absent from the left side of the navigation bar, because we add it with code at runtime.

Go to the Library, and find the icon for the Bar Button Item. This is different from the usual Round Rect Button, so make sure the object you've found lists its class as UIBarButtonItem. Drag the bar button to the right side of the navigation bar, where you'll find it automatically finds its way to a highlighted landing spot, making it the navigation bar's rightBarButtonItem. Select the bar button, bring up the Attributes inspector (⌘ 1), and change its identifier to Add. This will change its appearance to a simple plus sign (+).

The next step is to connect this button to the handleAddTapped method. This is a little different from the connections you've made thus far. First, when you bring up the button's Connections inspector (⌘ 2), you won't see the usual battery of touch events like Touch Up Inside. Instead, there's a single Sent Action called selector. This is because the UIBarButtonItem has a different object hierarchy than regular buttons and doesn't have UIControl, UIView, and UIResponder as superclasses. Instead, this object has properties called target and selector; when the bar button is tapped, the method named by selector is called on the target object. You could set both of those properties in code; since we're already here in Interface Builder, let's set it up here.

To set the selector and target, we drag the selector action from the Connections inspector to one of the other objects in the nib. This time, however, we *don't* drag it to the File's Owner. Since this is the MainWindow.xib, the File's Owner proxy object points to a generic UIApplication. The handleAddTapped method that we want the button to call is defined in the RootViewController class, so we drag the connection to the Root View Controller object in the nib window, as shown in Figure 5.6, on the next page. When you release the mouse button at the end of the

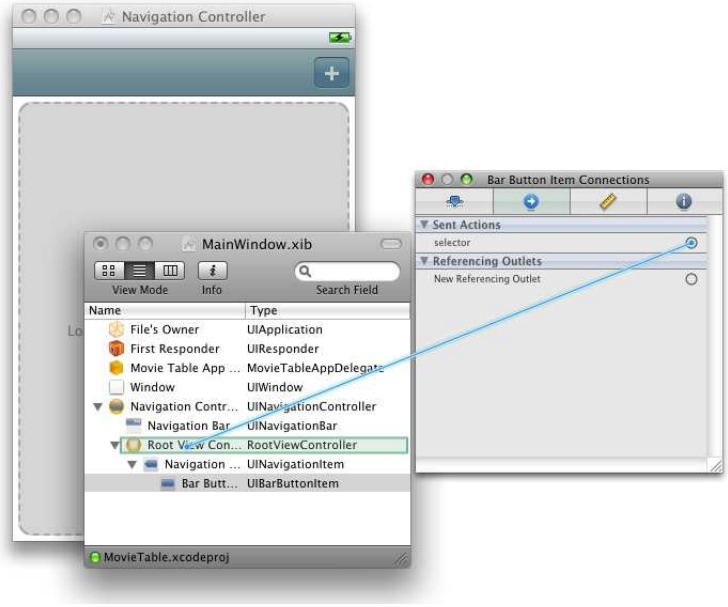

Figure 5.6: Connecting a UIBarButtonItem's selector to the RootViewController

drag, the names of the target's IBAction methods will appear, and you'll select the only one: handleAddTapped.

With the connection made, save in IB and return to Xcode. Now we can implement the handleAddTapped method that will be called when the user taps the Add button:

Download TableViews/MovieTable01/Classes/RootViewController.m

```
-(IBAction) handleAddTapped {
        Movie *newMovie = [[Movie alloc] init];
        editingMovie = newMovie;
        movieEditor.movie = editingMovie;
        [self.navigationController pushViewController:movieEditor animated:YES];
        // update UITableView (in background) with new member
        [moviesArray addObject: newMovie];
        NSIndexPath *newMoviePath =
                [NSIndexPath indexPathForRow: [moviesArray count]-1 inSection:0];
        NSArray *newMoviePaths = [NSArray arrayWithObject:newMoviePath];
        [self.tableView insertRowsAtIndexPaths:newMoviePaths withRowAnimation:NO];
        [newMovie release];
}
```

This method starts by creating an empty Movie object, setting it as the editingMovie, and navigating to the MovieEditorViewController, much like the code to edit an existing Movie did. What's different is that after navigating, it does cleanup work on the table view (while the table is out of sight) by adding the new object to the model array and then calling insertRowsAtIndexPaths:withRowAnimation: to update the table to reflect the new state of the model. The inserted Movie has blank fields, but when the user returns from the editor, the object will be updated in viewWillAppear:, just like when an existing item is edited.

Let's review. We used the navigation-application template to set up an application with a table view, which we backed with a model (a simple NSMutableArray) to provide a list of Movie objects. After looking at the various table cell styles, we added the ability to delete from the table either with horizontal swipes (by implementing tableView:canEditRowAtIndexPath:), or with the Edit button (by adding the default editButtonItem and implementing tableView:commitEditingStyle:forRowAtIndexPath:). Then we looked at how to access the UINavigationControl to navigate between view controllers and used the MovieEditorViewController to edit a Movie indicated by a selected row in the table and then to edit a new Movie in response to the tap of an Add bar button.

5.7 Custom Table View Cells

Back in Section 5.4, *Cell Styles*, on page 94, we looked at the four cell styles provided by iPhone OS. Although they suit a wide range of uses, sometimes you might want something else. If your GUI uses a unique color theme, the default black or blue text on white cells might not suit you. If you need to populate more than two labels, then none of the available styles will work for you.

It is possible, with a little work, to custom design your own table cell in Interface Builder and have your table use this design instead of the built-in styles. In this section, we'll use this technique to create a table that shows all three of the Movie fields.[5]

5. Because we will change so much in the project to use custom table cells, the book's downloadable code examples have split this exercise into a separate project. The previous material is represented by MovieTable01, and the custom-cell project is MovieTable02.

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

iPhone SDK Development's Home Page
http://pragprog.com/titles/amiphd
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/amiphd.

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)