User Interface Guidelines
and Database Best Practices

Pro iOS Table Views

for iPhone, iPad, and iPod touch

Tim Duckett

Apress®

Pro iOS Table Views for iPhone, iPad, and iPod Touch

Tim Duckett

Apress®

Pro iOS Table Views for iPhone, iPad, and iPod Touch

ISBN 978-1-4302-3348-0

ISBN 978-1-4302-3349-7 (eBook)

President and Publisher: Paul Manning
Lead Editor: Tom Welsh
Technical Reviewer: Brent Simmons
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Tracy Brown
Copy Editors: Valerie Greco, Sharon Wilkey
Compositor: MacPS, LLC
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`.

For Lucy, Kath, and Isaac

Contents at a Glance

Contents

About the Author

Tim Duckett designs and builds software with tools such as Objective-C and Ruby on Rails. Having been online since the early 1990s, he's worked with all kinds of clients in all kinds of sectors. Along the way he picked up an MBA and is a certified project manager, but asks that you don't hold those against him.

He lives in Sheffield in the UK with his family and far too many pets. In his spare time, he walks the dog, enjoys single malts, takes photos, and dismantles gadgets.

You can find him online at `http://adoptioncurve.net`, and on places including Twitter and GitHub as `timd`.

About the Technical Reviewer

Brent Simmons has been programming since his first Apple II Plus back in 1980. More recently he created TapLynx, a framework for creating iOS apps without programming. He also created the RSS reader NetNewsWire and the weblog editor MarsEdit. He's a cofounder of Sepia Labs, where he writes Glassboard for iOS. He blogs at `inessential.com`.

Acknowledgments

It's my name on the cover, but there's a whole host of people without whom this book would never have happened.

At Apress, Tracy Brown, Michelle Lowman, and Tom Welsh cracked the whip that got me from one end of the project plan to the other, and were very gracious about my procrastination and constant content changes. Sharon Wilkey and Valerie Greco unsplit all my infinitives. Brent Simmons was both kind and constructive as the technical reviewer, and if he laughed at my mistakes, he didn't tell me.

Aral Balkan and Sam Easterby-Smith were my metaphorical training wheels as I wobbled off to start my career as an Objective-C developer. And there are other people, too numerous to mention individually, who over the years who have provided support, inspiration, ideas, and source code. Tom Armitage is responsible for taking the only decent picture of me in existence.

Finally, none of this would have happened without the unconditional love and support of my family, who put up with all my grumbling and grumping while I wrote this book. I'm very lucky to have them.

Introduction

If you're an iOS app developer, chances are you'll be using table views somewhere in your development projects. Table views are the bread and butter of iOS apps. With them, you can create everything from the simplest of lists to fully tricked-out user interfaces.

Table views are one of the more complex components found in UIKit. Using them for (potentially boring!) standard user interfaces is quite simple, but customizing them can become much more challenging.

This book has a task-oriented focus to assist you when implementing customized table views. Although it delves deeply into the table view API, you can always choose the level of detail you want to dive into. This book aims to be a reference and customization cookbook at the same time, useful for beginners as well as intermediate developers.

What This Book Covers

Chapter 1, "Creating a Simple Table-View App," introduces the table view with some examples of the current state of the art. After showing you something of what's possible, we'll start out with a very simple table view–based app for the iPhone, which will introduce you to the `UITableView` and its main elements. The app will also act as a starting point for later versions, and it'll be a working prototype that you can use as the basis for your own experiments.

In **Chapter 2, "How The Table Fits Together,"** you'll look at how the parts of the table view work together. You'll see the main types of `UITableViews` and their anatomy. You'll learn how to create them both with Interface Builder and in code, and how to use the `UITableViewController` class as a template.

Chapter 3, "Feeding Your Tables With Data,"is about where the table gets its data and how you get it there. It shows how the table keeps track of sections and rows, and covers some of the software design patterns that the `UITableView` classes exploit.

Chapter 4 "How The Cell Fits Together," focuses on the cells that make up tables. You'll see how cells are structured internally, and how they're created and reused. It also covers the standard cells types that come for free with the `UITableView` classes.

Chapter 5, "Using Tables for Navigation," covers an almost-ubiquitous feature of the iOS user interface, and shows how tables can be used to navigate through a hierarchy of data in a simple and consistent way.

The constrained size of the iOS user interface presents some challenges when it comes to presenting large amounts of data. **Chapter 6, "Indexing, Grouping, and Sorting,"** presents some ways of arranging the data in tables, to help users find their way.

Chapter 7, "Selecting and Editing Table Content," shows how you can use tables to manage data. It covers how to add, delete, and rearrange the information, and some of the interface aspects that this entails.

In **Chapter 8, "Improving the Look of Cells,"** you will start to look at the process of going beyond standard cell types to customize the look and feel of your table views. This chapter covers two of the quickest ways to make the cells look the way you need them to.

Chapter 9, "Creating Custom Cells with Subclasses," takes customizing cells to the next level. You'll learn how to use custom `UITableViewCell` subclasses to gain detailed control over cells' appearance.

In addition to changing the look and feel of cells, you can make them truly interactive by embedding controls such as buttons and sliders. **Chapter 10, "Improving the Cell's Interaction,"** presents how to do this, as well as building cool table features such as slide-to-reveal, pull-to-refresh, and search.

Finally, in **Chapter 11, "Table Views on the iPad,"** you'll look at the iPad's split-view controller, which provides a flexible two-pane interface familiar from apps such as Mail.

The Style of This Book

I've tried to bridge the gap between two styles of book—the in-depth treatment of every last little detail, and the cookbook of specific point solutions. Both have their place, but sometimes I find that descriptions of very detailed, elegant solutions with lots of features can obscure the detail of the problem I'm trying to solve. Equally, sometimes cookbook solutions are too specific and don't easily lend themselves to adapting to my specific situation.

In the code examples that follow, I've tried to balance the two styles. The visual polish and extraneous functions are kept to a minimum, which hopefully results in examples that illustrate how to build a solution while also acting as a building block for your own code.

The Book's Source Code

You can download the source code for each chapter's examples from the Apress site or from GitHub at `http://github.com/timd/Pro-iOS-TableViews`.

Although that's the quickest way to get up and running, I encourage you to take the extra time to key in the code yourself as you go along. With Xcode's code completion, it doesn't take that long, and code that has flowed through your eyes and brain, and then out to your fingers, is much more likely to sink in and make sense.

Where to Find Out More

Beyond the pages of this book, there's a wealth of other information available online (not to mention the great range of other Apress titles):

- For a general overview, Apple's **"Table View Programming Guide for iOS"** is a detailed guide that covers most of the topics in this book. This is available online at `http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/TableView_iPhone/AboutTableViewsiPhone/AboutTableViewsiPhone.html`, or in Xcode's documentation.
- Apple's **iOS Developer Library** has full documentation for all Cocoa Touch libraries. It tends not to include examples in the documentation itself, but the Library is the one-stop shop for a detailed reference for each class, protocol, and library. Again, this is available online, at `http://developer.apple.com/library/ios/navigation/` or in Xcode's documentation library.
- **Online forums** are a fantastic resource. Sites such as Stack Overflow (`www.stackoverflow.com`) are the place to go for practical advice. Chances are, a number of people will have met and overcome the same problem that you're experiencing, and the answer will be there. Stack Overflow's customs and practices can be a little daunting at first, but it's worth persevering. There are no stupid questions, after all, just questions that haven't been answered yet.

- A general **Google search** will often throw up answers from blogs. There are some extremely talented individuals out there who regularly post about how to do this or that with iOS and Objective-C, and many of them also point to source code on their sites or GitHub and the like.
- Apple also provides some fairly detailed **source code examples**. Your mileage may vary with these. I sometimes find that they can be a bit overcomplicated and can obscure the core technique that I'm trying to grasp. But they shouldn't be overlooked, if only because they've been written by engineers with an intimate understanding of the frameworks.
- **Universities such as Stanford and MIT** place entire semesters' worth of lecture modules online, both on their sites and on iTunes U. Their technical education is some of the best on the planet, and some of the online lectures are taught by Apple engineers. These are definitely worth checking out.
- **Local user groups**, including CocoaDev, CocoaHeads, and NS$city$ (where *$city$* is a location near you), are groups that meet regularly around the world. It's an iron law of software that there's always someone who knows more than you do about a topic, and problems are always less daunting when discussed with them over a beer or two.
- **Mailing lists** such as cocoa-dev don't perhaps have the profile of sites such as Stack Overflow these days, but can still be a useful resource. An excellent example (which covers not just coding topics, but design and business issues as well) is `http://iosdevweekly.com`.

Finally, if you've battled with—and resolved—some gnarly issue, then *post about it yourself*, whether that's on your own blog or a site like Stack Overflow. Even if the topic has been covered numerous times before, there's always room for another take on a problem. Your unique point of view could be just what someone else needs.

Contacting the Author

Tim Duckett can be found online at `http://adoptioncurve.net` and on Twitter as `@timd`.

Chapter 1

Table Views from the Ground Up

In this chapter, you'll start your exploration of table views. It begins with an overview of what table views are and some examples of how they're used in practice. Then in the second section, you'll build a simple "Hello, world"-style table view app to introduce you to the components behind the user interface and help you to contextualize the detail that's going to come in later chapters.

If you're just starting to use table views, it's worth taking some time to build a very simple one from scratch before diving into the gnarly details. However, if you've reached the stage where you feel more confident about how the components of the table view jigsaw fit together and want to get straight into the code, feel free to skip the rest of this chapter completely. I'll cover the elements in detail later, so you won't miss out.

What Are Table Views?

Examples of table views are to be found everywhere in iOS apps. You are already familiar with simple tables, implemented as standard controls such as the iPhone's Contacts app or the iPad's Mail app, shown in Figure 1–1.

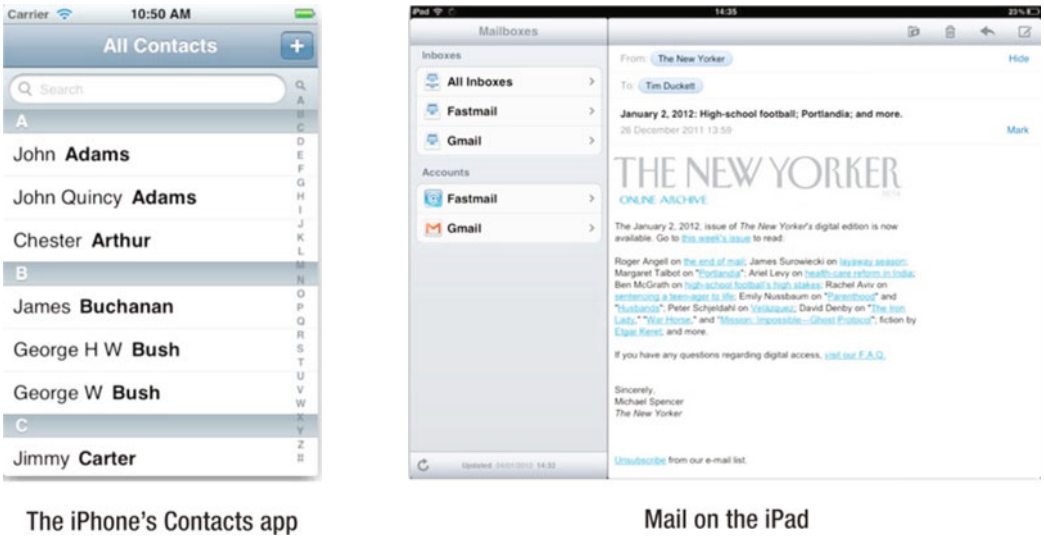

Figure 1–1. *Some basic table-based applications*

At the other end of the scale, the default look, feel, and behavior of the table view and cells can be customized to the point where they are hardly recognizable as table views at all. Figure 1–2 shows some examples.

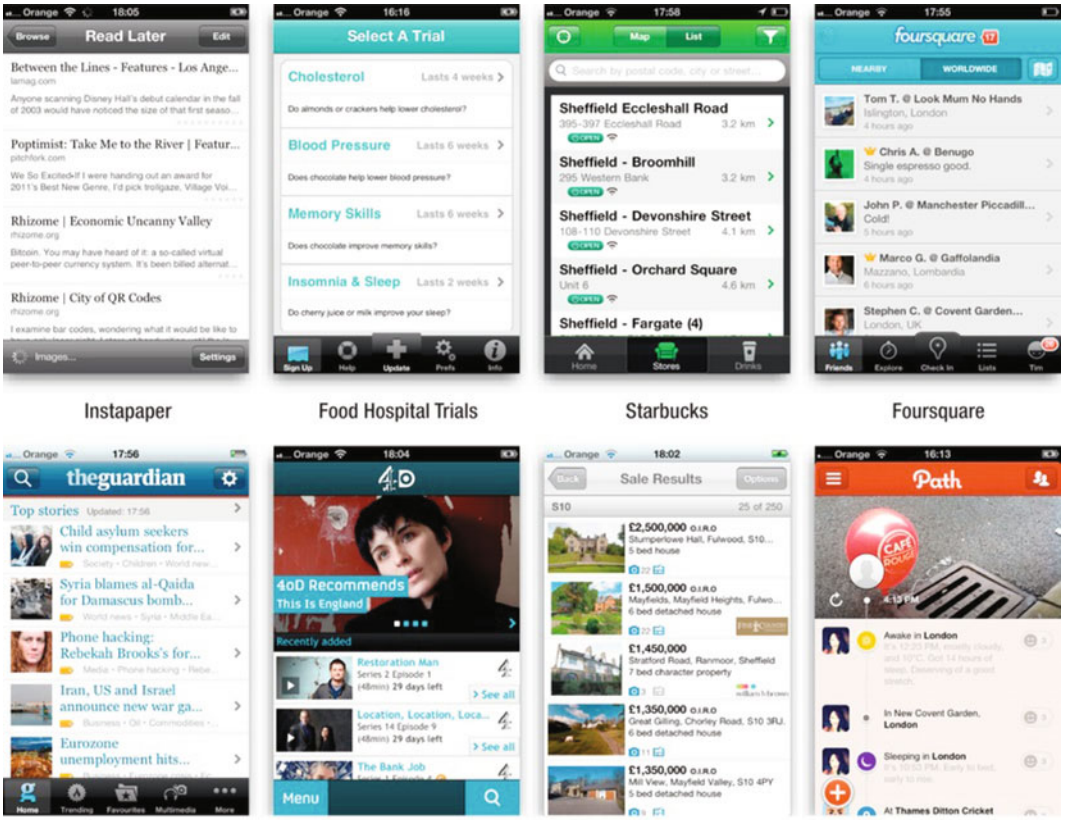

Figure 1–2. *Examples of table views in action on the iPhone*

On the iPad, table views are often used as components of a larger user interface. Figure 1–3 shows an example from the wildly successful (and addictive!) Carcassonne game.

Figure 1–3. *Two table views in action on the iPad*

The Anatomy of a Table View

The table view displays a list of elements—or cells—that can be scrolled up and down vertically. They are instances of the `UITableView` class and come in two physical parts:

- The container part—the `tableView` itself—is a subclass of `UIScrollView` and contains a vertically scrollable list of table cells.
- Table cells, which can either be instances of one of four standard `UITableViewCell` types or custom subclasses of `UITableViewCell` that can be customized as required.

Figure 1–4 illustrates the parts of a table view.

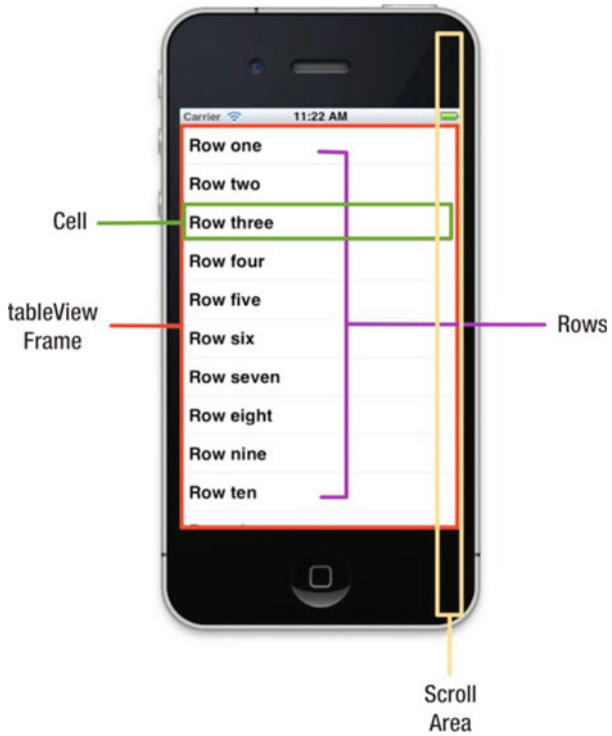

Figure 1–4. *The basic anatomy of a table view*

Table view operations are supported by two `UITableView` protocols:

- `UITableViewDatasource` provides the table view with the data that it needs to construct and configure itself, as well as providing the cells that the table view displays.
- `UITableViewDelegate` handles most of the methods concerned with user interaction, such as selection and editing.

Creating a Simple Table View App

In the rest of this chapter, you'll build a simple "Hello, world"-style table view app from scratch. It will show you how the container, cells, data source, and delegate all fit together and give you an app that you can use as the basis for your own experiments.

I'm going to take it deliberately slowly and cover all the steps. If you're a confident Xcode driver, you won't need this hand-holding—just concentrate on the code instead.

Still with me? Okay—you're going to do the following:

- Create a simple, window-based application skeleton
- Generate some data for feeding the table

- Create a simple table view
- Wire up the table view's data source and delegate
- Implement some very simple interactivity

It's all very straightforward but useful practice. Onward!

Creating the Application Skeleton

For this application, you're going to use a simple structure: a single view managed by a view controller, and a NIB file to provide the content for the view. Fire up Xcode and select the Single View Application template, as shown in Figure 1–1.

> **NOTE:** With each new release of Xcode, Apple frequently (and pointlessly) changes the templates that are included. You may see a set that is different from those shown in Figure 1–5. Check the description of the templates to find the one that will provide a single-view application.

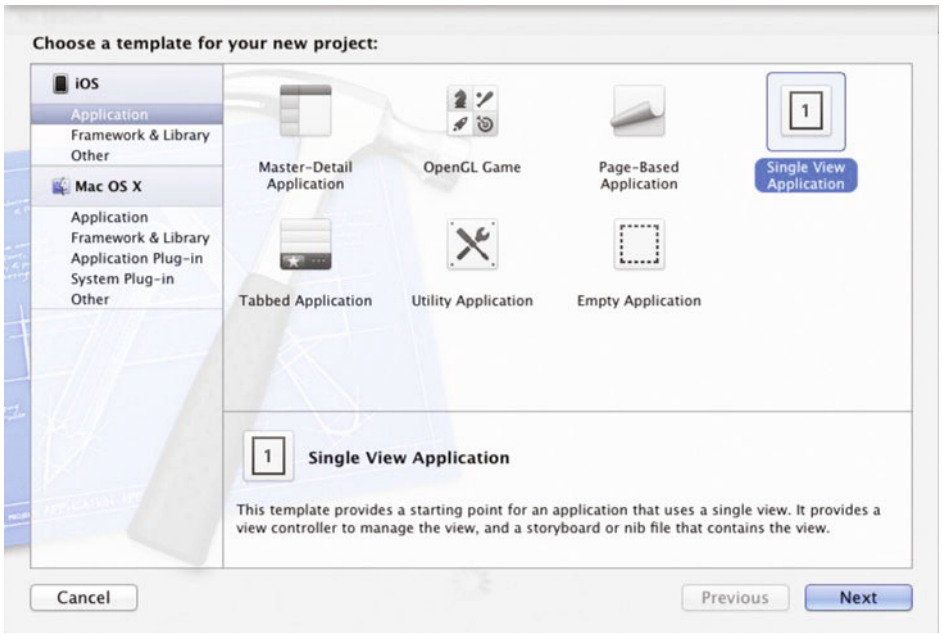

Figure 1–5. *Xcode's template selection pane*

Call the application **SimpleTable**. You're going to build an iPhone version. You don't need the storyboard or unit tests, but you do want automatic reference counting. Make sure those options are selected as needed, as shown in Figure 1–6.

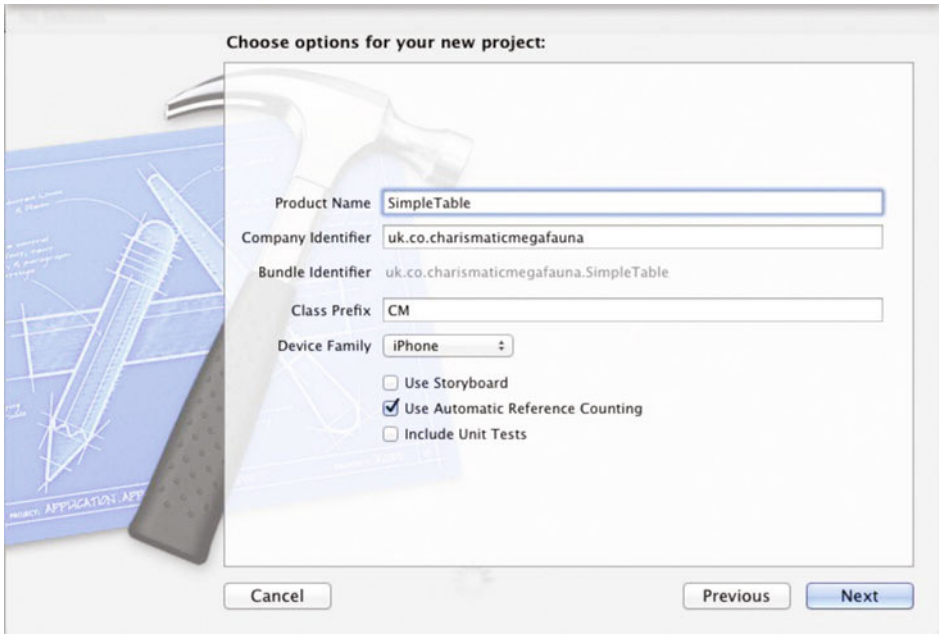

Figure 1–6. *Name the application.*

The Class Prefix setting will be prefixed to the names of any classes that you create within the application—so in this case, the `AppDelegate` class would be named `CMAppDelegate`. This prevents any chance of you inadvertently creating classes that clash with existing ones, either in the iOS SDK or any libraries that you might be using.

The choice of the class prefix is entirely up to you. I tend to use CM for Charismatic Megafauna, but you could use something along the lines of ST for Simple Table instead.

Finally, you'll need to select a location to save the project to. You don't need to worry about creating a local Git repository for this project unless you particularly want to.

When you've reached this point, you'll see the project view of Xcode, with the initial skeleton of your application. It'll look something like Figure 1–7, assuming that you've stuck with the SimpleTable application name.

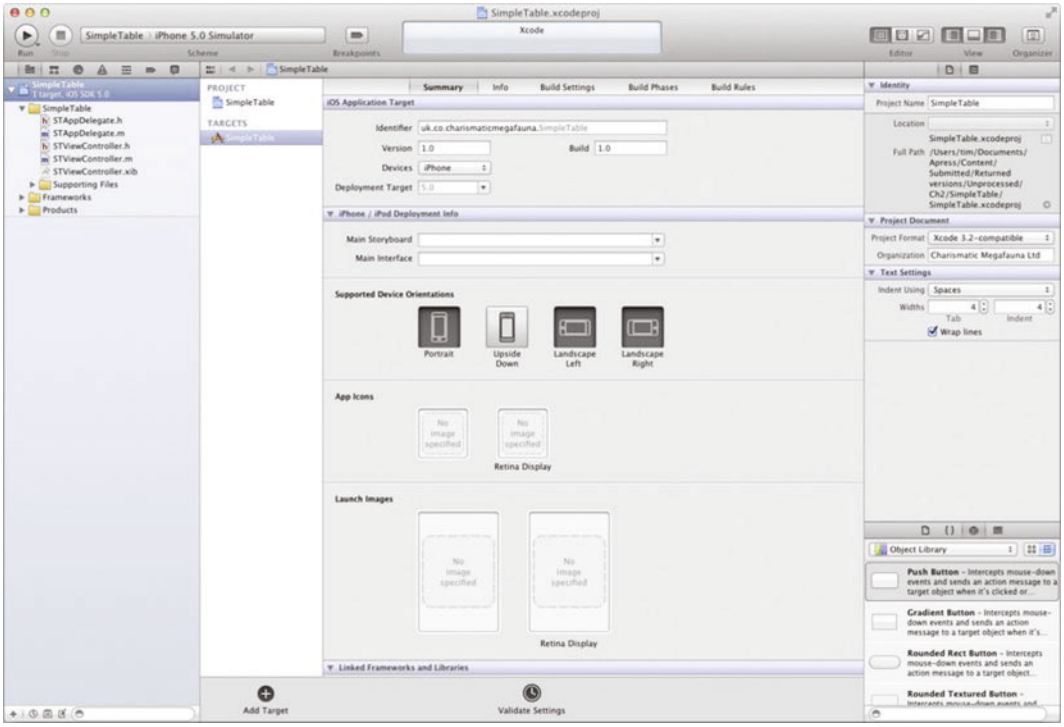

Figure 1–7. *The initial Xcode view showing our new skeleton application*

You'll see that you have the following:

- An app delegate (`STAppDelegate.h` and `.m`)
- A table-view controller (`STViewController.h` and `.m`)
- A NIB file containing the view (`STViewController.xib`)

At the end of this chapter, you'll look again at how these fit together. For the moment, you'll be working with the table view controller and its NIB file.

Generating Some Data

Before you start with the table view itself, you need to create some data to feed it. Because this is a simple table example, the data is going to be simple too. You'll create an `NSMutableArray` of `NSStrings` that contains some information to go into each cell.

The data array will need to be ready by the time the data source is called by the table view—so where to create it? There are several options, but one obvious place is in the view controller's `viewDidLoad` method. This method gets called the first time the view controller is loaded, which takes place well before the table will try to populate itself, so you'll be safe to create it here.

You're also going to need a way of passing the array of data around the application. This process requires a property that can be accessed by the various methods that will need access to the data.

Let's get started. Open the `STViewController.h` file (shown in Figure 1–8) and begin by creating the property.

> **NOTE:** To save space from now on, I'm not going to show the full Xcode interface—just the code that you'll need to enter.

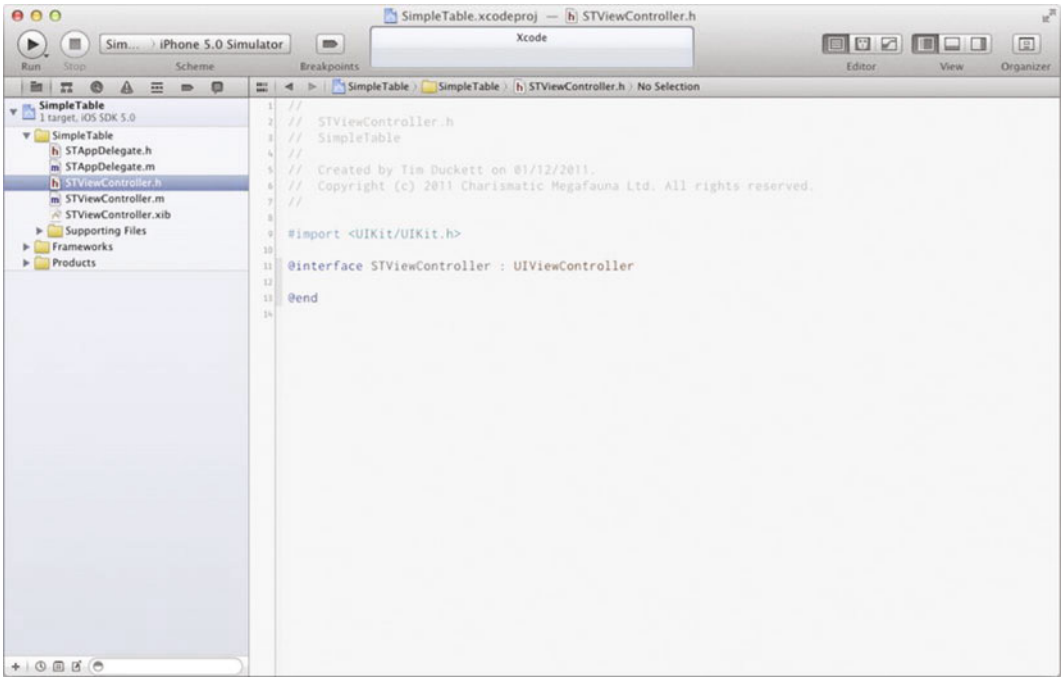

Figure 1–8. *Editing the `STViewController.h` file*

Add in a declaration for the property so that the code looks like Listing 1–1.

Listing 1–1. *Declaring the Property*

```
#import <UIKit/UIKit.h>

@interface STViewController : UIViewController

@property (nonatomic, strong) NSMutableArray *tableData;  // holds the table data

end
```

Save this file and then switch to the implementation file.

> **TIP:** In Xcode 4, you can flip quickly between the `.h` and `.m` files with the Ctrl + Command + Up Arrow key combination, or by clicking the filename in the breadcrumb display at the top of the code editor then selecting the file from the drop-down menu that appears.

First, you'll need to synthesize the `tableData` property. Add the following line after `@implementation STViewController`:

```
    @synthesize tableData;
```

Now you're going to create the actual data array itself. In the `View lifecycle` section of the `STViewController.m` file, you'll need to find the `viewDidLoad` method.

Our data array will be a simple array of ten `NSStrings`, each with a number. Add the code shown in Listing 1–2.

Listing 1–2. *Creating the Data Array*

```
- (void)viewDidLoad
{
    // Run the superclass's viewDidLoad method
    [super viewDidLoad];

    // Create the array to hold the table data
    self.tableData = [[NSMutableArray alloc] init];

    // Create and add 10 data items to the table data array
    for (NSUInteger i=0; i < 10; i++) {

        // The cell will contain a string "Item X"
        NSString *dataString = [NSString stringWithFormat:@"Item %d", i];

        // Here the new string is added to the end of the array
        [self.tableData addObject:dataString];

    }

    // Print out the contents of the array into the log
    NSLog(@"The tableData array contains %@", self.tableData);

}
```

Having created the `tableData` array in the `viewDidLoad` method, you'll need to clean up after yourself. There's no right or wrong place to do this. However, I've developed a habit: if I create an object in one method that needs to persist, I try to put the cleanup code in a corresponding method.

Because you create the `tableData` array in the `viewDidLoad` method, you can clean it up in the `viewDidUnload` method. That's what the code in Listing 1–3 does.

Listing 1–3. *Cleaning Up*

```
- (void)viewDidUnload
{
    [super viewDidUnload];

    self.tableData = nil;
}
```

Let's run the application to see that data array being created. The user interface isn't much to write home about yet, but you'll be able to see the data that you're going to feed to the table.

Run the application in the Simulator by pressing Command + R or by choosing **Product ➤ Run**, and then take a look at the logger output:

```
YYYY-MM-DD HH:MM:SS.sss SimpleTable[21502:ef03] The tableData array contains (
    "Item 0",
    "Item 1",
    "Item 2",
    "Item 3",
    "Item 4",
    "Item 5",
    "Item 6",
    "Item 7",
    "Item 8",
    "Item 9"
)
```

A NOTE ABOUT MEMORY MANAGEMENT IN IOS 5

Newcomers to iOS often arrive having heard hair-raising tales of the hideous complexity of memory management in Objective-C. There's a school of thought that says that all "modern" languages should have baked-in memory management such as garbage collection, and managing memory manually is somehow old-fashioned, difficult, and a waste of time. Someone in Apple was obviously listening to that school of thought, because perhaps the single most-anticipated feature of iOS 5 was automatic reference counting (ARC).

Put simply, ARC uses some compiler magic to take care of all the memory management issues that you had to worry about in previous versions. Prior to iOS 5, in order to prevent memory leaks, you had to "balance" any `alloc`, `retain` or `copy` of an object with a corresponding `release` (or `autorelease`). Miss a release, and you had a potential memory leak. Over-release, and you risked crashing your app by sending messages to deallocated objects. Having ARC means this is no longer something you need to worry about; switch it on in the compiler, and memory management will be taken care of "automagically."

NOTE: Automatic reference counting is supported in only iOS 5 and Xcode versions 4.2 and above.

Creating the Table View

As it stands, the user interface for our application is a bit dull. You haven't added the table yet! This needs fixing.

Click the `STViewController.xib` file in the project explorer, and you'll see the NIB file open in the Interface Builder pane, as shown in Figure 1–9.

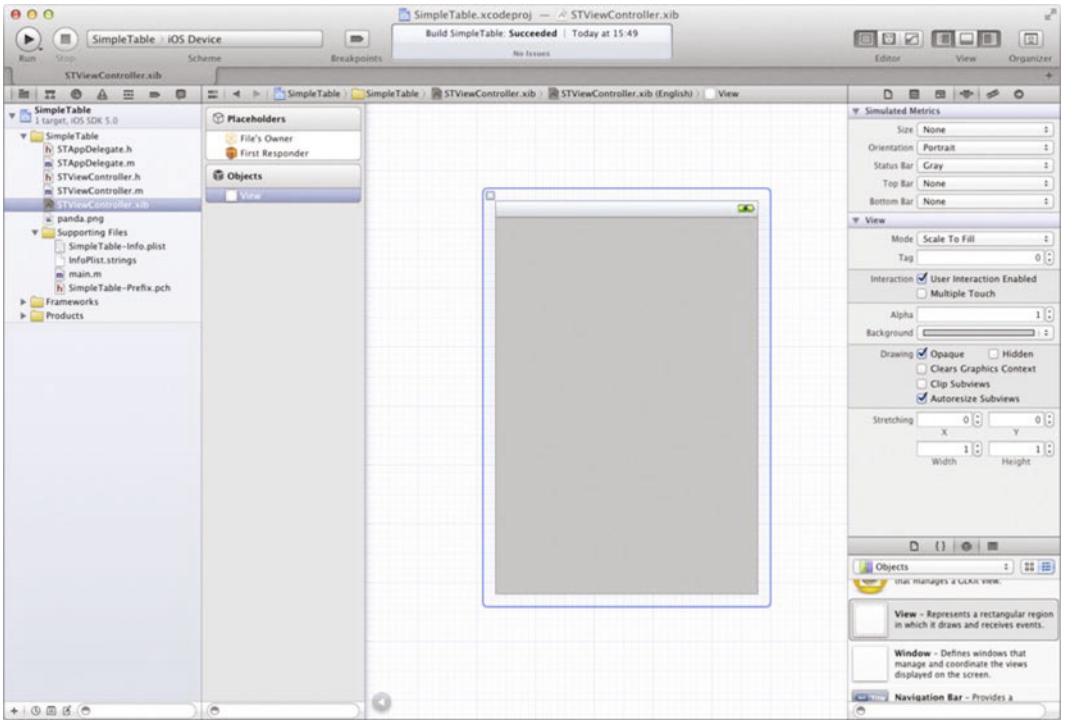

Figure 1–9. *Editing the NIB file in Interface Builder*

In the Objects browser at the bottom right, find the *Table View* item and drag it out onto the view in the center. By default, the table view will try to expand to fit the full view, but you need to resize it by grabbing the resize handles and making it smaller. Finally, grab a *Label* from the object browser and drop that onto the top of the view so it looks like Figure 1–10.

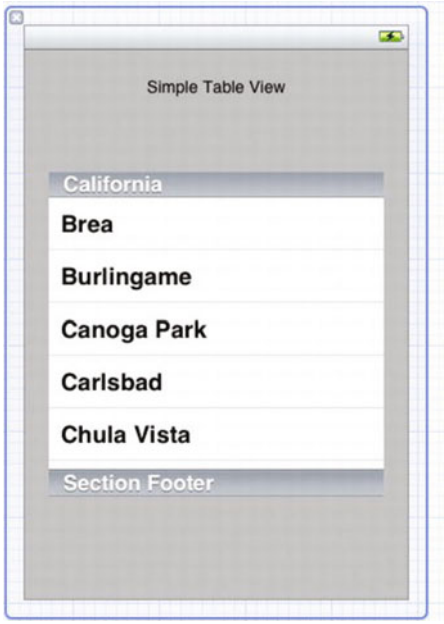

Figure 1–10. *Resizing the table view*

Believe it or not, that's all you need to do in order to implement the most basic table view. It won't display any data yet, because you haven't implemented the `UITableViewDataSource` protocol methods, and it certainly won't have any interactivity—but the app will run.

To prove this, run it again (Command + R), and marvel at our awesome application in the Simulator, as shown in Figure 1–11.