PART I

Manipulating and Displaying Data on the iPhone and iPad

1 Introducing Data-Driven Applications

WHAT'S IN THIS CHAPTER?

- Creating a view-based application using Xcode
- Building a simple data model
- Neatly displaying your data in a table using the `UITableView` control

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/remtitle.cgi?isbn=1118391845` on the Download Code tab. The code is in the Chapter 1 download and individually named according to the names throughout the chapter.

Data is the backbone of most applications. It is not limited only to business applications. Games, graphics editors, and spreadsheets all use and manipulate data in one form or another. One of the most exciting things about iOS applications is that they enable you and your customers to take your data anywhere. The mobility of iOS devices gives the developer an amazing platform for developing applications that work with that data. You can use the power of existing data to build applications that use that data in new ways. In this book, you will learn how to display data, create and manipulate data, and send and retrieve data over the Internet.

In this chapter, you learn how to build a simple data-driven application. While this application will not be production-ready, it is intended to help you to get started using the tools that you will use as you build data-driven applications. By the end of this chapter, you will be able to create a view-based application using Xcode that displays your data using the `UITableView` control. You will also gain an understanding of the Model-View-Controller (MVC) architecture that underlies most iOS applications.

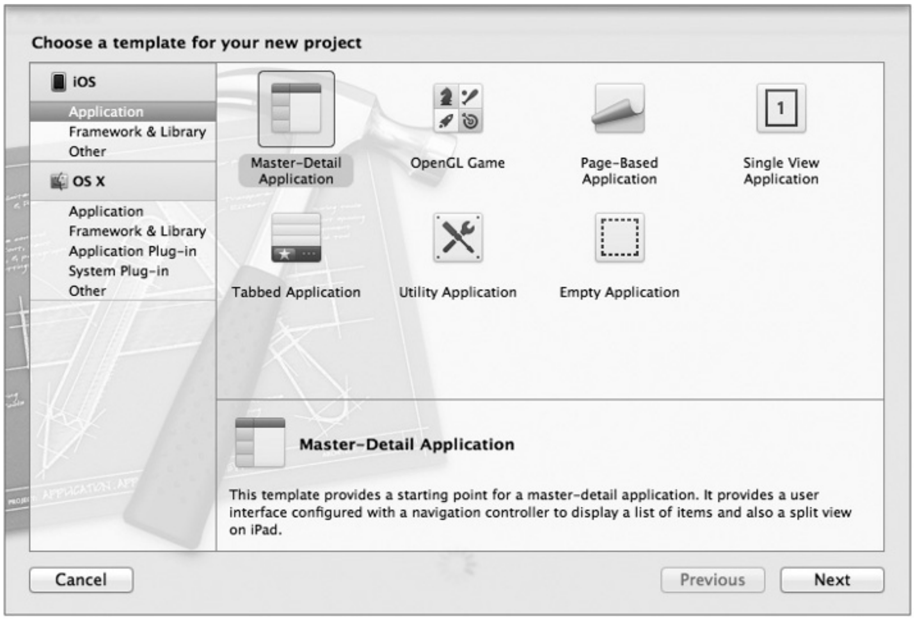
BUILDING A SIMPLE DATA-DRIVEN APPLICATION

Many applications that you will build for iOS will need to handle data in one form or another. It is common to display this data in a table. In this section, you learn how to build an application that displays your data in a table in an iOS application.

Creating the Project

To build applications for iOS, you need to use the Xcode development environment provided by Apple. Xcode is a powerful integrated development environment that has all of the features of a modern IDE. It integrates many powerful features including code editing, debugging, version control, and software profiling. If you do not have Xcode installed, you can install it via the Mac App Store or by downloading it from the Apple developer website at `https://developer.apple.com/xcode/index.php`.

To begin, start up Xcode and select File ➪ New ➪ Project. A dialog box appears that displays templates for the various types of applications that you can create for iOS and Mac OS X, as shown in Figure 1-1.



This template provides a starting point for a master-detail application. It provides a user interface configured with a navigation controller to display a list of items and also a split view on iPad.

FIGURE 1-1: New Project dialog

Each option presented provides you with the basic setup needed to start developing your application. The default iOS templates are divided into three groups: Application, Framework & Library, and Other.

The Application group consists of the following templates:

- **Master-Detail Application:** This template provides a starting point for a master-detail application. It provides a user interface configured with a navigation controller to display a list of items and a split view on iPad. The Contacts application is an example of a Master-Detail Application. The contact list is the master table and detail is the individual contact information that you see when you tap a contact.
- **OpenGL Game:** This template provides a starting point for an OpenGL ES-based game. It provides a view into which you render your OpenGL ES scene and a timer to allow you to animate the view. OpenGL is a graphics language that you can use to build games and other graphics-intensive applications. You will not be using OpenGL in this book.
- **Page-Based Application:** This template provides a starting point for a page-based application that uses a page view controller. You can use this template to build applications that incorporate page layout and page turning animations like iBooks.
- **Single View Application:** This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view and a storyboard or nib file that contains the view. The calculator application is an example of a single view application.
- **Tabbed Application:** This template provides a starting point for an application that uses a tab bar. It provides a user interface configured with a tab bar controller, and view controllers for the tab bar items. The Clock application is a tabbed application. The tabs at the bottom let you switch between the world clock, alarm, stopwatch, and timer.
- **Utility Application:** This template provides a starting point for a utility application that has a main view and an alternate view. For iPhone, it sets up an Info button to flip the main view to the alternate view. For iPad, it sets up an Info bar button that shows the alternate view in a popover. Weather is a utility application. It provides a simple interface and an information button that flips the interface to allow for advanced customization.
- **Empty Application:** This template provides a starting point for any application. It provides just an application delegate and a window. You can use this template if you want to build your application entirely from scratch without very much template code.

The Framework & Library template set consists of a single template: Cocoa Touch Static Library. You can use this template to build static libraries of code that link against the Foundation framework. You will not use this template in this book. However, this template is useful for building libraries of code that you can share amongst multiple projects.

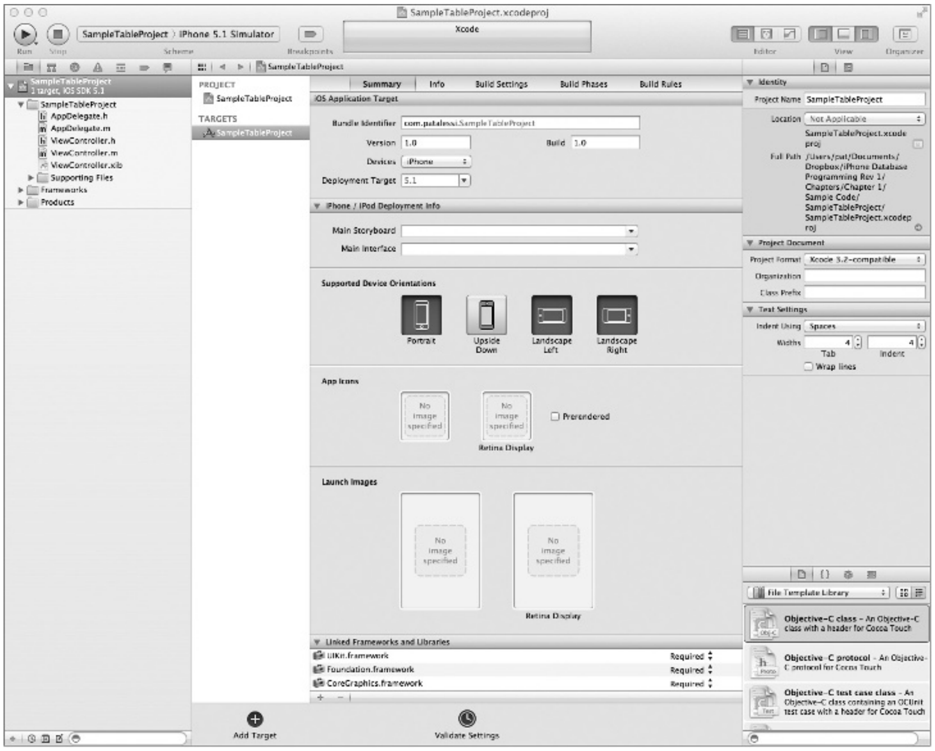
The Other template set consists of only a single template: Empty. The Empty template is just an empty project waiting for you to fill it up with code. You will not use this template in this book, but it is useful if you want to start a new project and none of the existing templates is appropriate.

For this sample application, you are going to use the straightforward Single View Application template.

Select the Single View Application template from the dialog box and click the Next button. Set the Product Name of your project to **SampleTableProject**, leave the Company Identifier at its default setting, and choose iPhone from the Devices drop-down. Check the checkbox labeled Use Automatic

Reference Counting, and make sure that the other two checkboxes are unchecked. Click the Next button. Select a location to save your project, and click the Create button. Xcode creates your project and presents the project window. You are now ready to get started!

Xcode now displays the project window, as shown in Figure 1-2. In the project window, you will see the navigator in the left-hand pane. The navigator simplifies navigation through various aspects of your project. The selector bar at the top of the navigator area allows you to select the specific area that you want to navigate. There are seven different navigators that you can use.



iOS Application Target

Bundle Identifier: com.patalessi.SampleTableProject
Version: 1.0
Build: 1.0
Devices: iPhone
Deployment Target: 5.1

iPhone / iPod Deployment Info

Main Storyboard:
Main Interface:

Supported Device Orientations: Portrait, Upside Down, Landscape Left, Landscape Right

App Icons: No image specified; No image specified (Retina Display); Prerendered

Launch Images: No image specified; No image specified (Retina Display)

Linked Frameworks and Libraries

UIKit.framework	Required
Foundation.framework	Required
CoreGraphics.framework	Required

Identity

Project Name: SampleTableProject
Location: Not Applicable
SampleTableProject.xcodeproj
Full Path: /Users/pat/Documents/Dropbox/iPhone Database Programming Rev 1/Chapters/Chapter 1/Sample Code/SampleTableProject/SampleTableProject.xcodeproj

Project Document

Project Format: Xcode 3.2-compatible
Organization:
Class Prefix:

Text Settings

Indent Using: Spaces
Widths: 4 (Tab) 4 (Indent)
Wrap lines

FIGURE 1-2: Xcode project window

The first icon in the navigator selector bar represents the Project Navigator. You use the Project Navigator to view the files that are part of your project. You can organize your files in folders and use the Project Navigator to drill down through these folders to get to the file that you want to edit.

Whatever item you select in the Project Navigator appears in the Editor area. Selecting a source code file (`.m` extension) will result in that file opening in the Editor area. If you select a user interface file (`.xib` extension), Interface Builder will open in the Editor area, allowing you to work on the user interface file. Double-clicking on a code file opens the code in a new tab, which can make your code easier to work with. If you prefer that double-clicking a code file opens the code in a new window, you can change this behavior in the General tab of Xcode ➪ Preferences dialog.

You can change various configuration settings for your project by selecting the project node at the top of the Project Navigator to open the project settings in the Editor area.

In addition to the Project Navigator, you can also find the Symbol, Search, Issue, Debug, Breakpoint, and Log Navigators in the left pane. You can use these various navigators to get different views of your project. Feel free to click on any of the folders, files, or navigator tab icons to explore how Xcode organizes your project. You can add additional folders at any time to help keep your code and other project assets such as images, sound, and text files under control.

Adding a UITableView

The most common control used to display data on iOS is the `UITableView`. As the name suggests, the `UITableView` is a view that you can use to display data in a table. You can see the `UITableView` in action in iOS's Contacts application. The application displays your list of contacts in a `UITableView` control. You learn much more about the `UITableView` control in Chapter 3.

Typically, when developing the interface for your iOS applications, you will use the Interface Builder. This tool is invaluable for interactively laying out, designing, and developing your user interface. However, the focus of this chapter is not on designing a beautiful interface; it is on displaying data on iOS. So instead of using Interface Builder to design the screen that will hold a table view, you will just create and display it programmatically.

To create the table view, you will be modifying the main View Controller for the sample project.

Model-View-Controller Architecture

Before I move on with the sample, it's important that you understand the basic architecture used to build most iOS applications: Model-View-Controller. There are three parts to the architecture, shown in Figure 1-3. As you can probably guess, they are the model, the view, and the controller.

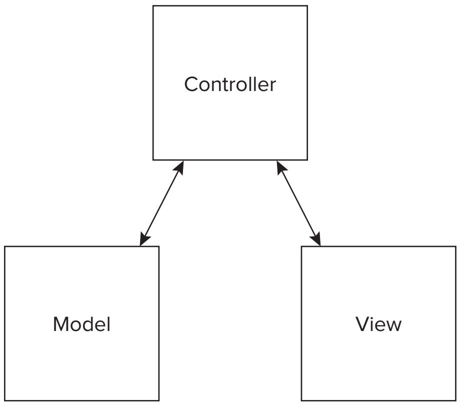


FIGURE 1-3: Model-View-Controller architecture

The *model* is the class or set of classes that represent your data. You should design your model classes to contain your data, the functions that operate on that data, and nothing more. Model classes should not need to know how to display the data that they contain. In fact, think of a model class as a class that doesn't know about anything else except its own data. When the state of the data in the model changes, the model can notify anyone interested, informing the listener of the state change. Alternatively, controller classes can observe the model and react to changes.

In general, model objects should encapsulate all your data. Encapsulation is an important object-oriented design principle. The idea of encapsulation is to prevent other objects from changing your object's data. To effectively encapsulate your data, you should implement interface methods or properties that expose the data of a model class. Classes should not make their data available through public variables.

A complete discussion of object-oriented programming is beyond the scope of this book, but there are many good resources for learning all about OOP. I have included some sources in the "Further Exploration" section at the end of this chapter. Encapsulating your data in model objects will lead to good, clean, object-oriented designs that you can easily extend and maintain.

The *view* portion of the MVC architecture is your user interface. The graphics, widgets, tables, and text boxes present the data encapsulated in your model to the user. The user interacts with your model through the view. View classes should contain only the code that is required to present the model data to the user. In many iOS applications, you won't need to write any code for your view. Quite often, you will design and build it entirely within Interface Builder.

The *controller* is the glue that binds your model to your view. The controller contains all the logic for telling the view what to display. It is also responsible for telling your model how to change based on input from the user. Almost all the code in your iOS applications will be contained in controller classes.

To quickly summarize, the *model* is your application data, the view is the user interface, and the controller is the business logic code that binds the view to the model.

In the sample application, you will be creating a class that acts as the model for the data. Xcode creates a default controller for the application as part of the Single View Application code template, and you are going to add a `table view` as the view so that you can see the data contained in the model. You will then code the controller to bind the view and model.

Adding the Table View Programmatically

Now that you have a basic understanding of the MVC architecture, you can move ahead and add the `table view` to the application. Open the SampleTableProject folder in the Project Navigator pane by clicking the disclosure triangle to the left of the folder. Select the `ViewController.m` file to display its code in the code window.

Because you are not using Interface Builder to build the interface for this application, you will override the `loadView` method of the `UIViewController` to build the view. You are going to write code in the `loadView` method so that when the View Controller tries to load the view, it will create an instance of the `UITableView` class and set its `view` property to the newly created view. Add the following code to the `ViewController` class implementation:

```
- (void)loadView {
    CGRect cgRct = CGRectMake(0, 20, 320, 460);
    UITableView * myTable = [[UITableView alloc] initWithFrame:cgRct];
    self.view = myTable;
}
```

The first line creates a `CGRect`, which is a Core Graphics structure used to specify the size and location for the `table view`. You set it to have its origin at (0, 20), and defined it to be 320 points wide by 460 points high. The `table view` will cover the entire screen, but start 20 points from the top, below the status bar.

The next line creates an instance of a `UITableView` and initializes it with the dimensions that you specified in the previous line.

Just creating the `table view` instance is not enough to get it to display in the view. You have to inform the View Controller about it by setting the View Controller's `view` property to the `table view` that you just created.

You can go ahead and click the Run icon in the toolbar at the top left of the Xcode window to run the project. You will see that your application compiles successfully and starts in the iOS simulator. You should also see a bunch of gray lines in the simulator, as shown in Figure 1-4. That is your `table view`! Those gray lines divide your rows of data. Unfortunately, there is no data for your `table view` to display yet, so the table is blank. You can, however, click in the simulator and drag the `table view`. You should see the lines move as you drag up and down and then snap back into position as you let go of the mouse button.

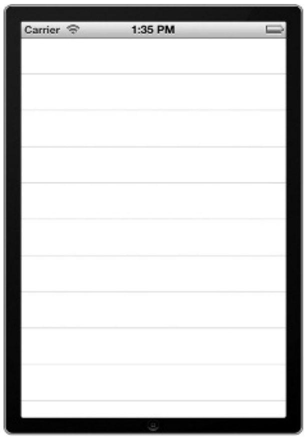


FIGURE 1-4: Running an application with table view

Retrieving Data

A table is useless without some data to display. To keep this first example simple, you are going to create a very simple data model. The data model will be a class that contains a list of names that you would like to display in the table. The model class will consist of an array to hold the list of data, a method that will return the name for any given index, and a method that will return the total number of items in the model.

To create a new class in your project, begin by clicking the SampleTableProject folder in the Project Navigator in Xcode. Then select File ➪ New ➪ File. You will see the New File dialog that shows all of the types of files that you can create in an Xcode project.

Select Cocoa Touch in the left pane of the dialog box, select Objective-C Class as the type of file that you want to create, and click Next. On the next screen, name your class `DataModel` by typing **DataModel** into the Class text box. Then select `NSObject` in the drop-down box next to Subclass Of. The template allows you to create Objective-C classes that are subclasses of `NSObject`, `UIView`, `UIViewController`, `UITableViewController`, or `UITableViewCell`. In this case, you want to create a subclass of `NSObject`, so just leave `NSObject` selected. Click Next to move to the net screen.

In the final dialog screen, you can tell Xcode where to put your new file. The options in this dialog allow you to specify the location for your file, the group to contain your new class, and the build target to compile your file. Leave the options at their defaults and click Create to have Xcode generate your new class files.

Implementing Your Data Model Class

For your class to serve data to the `table view`, you'll need a method to return the requested data. So, you'll create an interface method called `getNameAtIndex` that will return the name from the model that corresponds with the index that is passed in.

Bring up your `DataModel.h` header file by selecting it in the Project Navigator in Xcode. Below the interface definition, add the following line of code to declare the `getNameAtIndex` interface method:

```
-(NSString*) getNameAtIndex:(int) index;
```

You will also need an interface method that tells users of the class how many rows you will be returning. So, add another method to the interface called `getRowCount`. Below the declaration for `getNameAtIndex`, add the declaration for `getRowCount`:

```
-(int) getRowCount;
```

Your header file should look like this:

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

@interface DataModel : NSObject
-(NSString*) getNameAtIndex:(int) index;
-(int) getRowCount;

@end
```

Now switch over to the data model implementation file `DataModel.m` and implement the new methods. You can quickly switch between a header and an implementation file in Xcode by using the shortcut key combination Ctrl+Cmd+Up Arrow. You can also use the Assistant Editor to view your source and header files side by side. The Assistant Editor is the second icon in the Editor menu bar on the top right side of the Xcode window. It looks like a tuxedo.

NOTE *You will be well served to learn the keyboard shortcuts in Xcode. The small amount of time that you invest in learning them will more than pay off in time saved.*

Below the `#import` statement in your implementation file, add a local variable to hold the data list. Typically, the data in your application will come from a database or some other datasource. To keep this example simple, you'll use an `NSArray` as the datasource. Add the following line to the `DataModel.m` file below the `#import` statement:

```
NSArray* myData;
```

Now, inside the `@implementation` block, you add the implementation of the `getNameAtIndex` method. Add the following code stub between the `@implementation` and `@end` tags in `DataModel.m`:

```
-(NSString*) getNameAtIndex:(int) index
{

}
```

Before you get to the lookup implementation, you need to initialize the data store, the `myData` array. For this example, you do that in the initializer of the class. Above the function stub `getNameAtIndex`, add the following code to initialize the class:

```
-(id)init
{
    if (self = [super init])
    {
        // Initialization code
        myData = [[NSArray alloc] initWithObjects:@"Albert", @"Bill", @"Chuck",
                  @"Dave", @"Ethan", @"Franny", @"George", @"Holly", @"Inez",
                  nil];
    }
    return self;
}
```

The first line calls the superclass's `init` function. You should always call `init` on the superclass in any subclass that you implement. You need to do this to ensure that attributes of the superclass are constructed before you begin doing anything in your subclass.

The next line allocates memory for the array and populates it with a list of names.

The final line returns an instance of the class.

Now that you have the data initialized, you can implement the function to get your data. This is quite simple in this example. You just return the string at the specified location in the array like so:

```
-(NSString*) getNameAtIndex:(int) index
{
    return (NSString*)[myData objectAtIndex:index];
}
```

This line of code simply looks up the object at the specified index in the array and casts it to an `NSString*`. You know that this is safe because you have populated the data by hand and are sure that the object at the given index is an `NSString`.

NOTE *To keep this example simple, I have omitted bounds checking that you would add in a production application.*

To implement `getRowCount`, you simply return the count of the local array like this:

```
-(int) getRowCount
{
    return [myData count];
}
```

At this point, if you build your project by selecting Product ➪ Build from the menu or pressing Cmd+B, your code should compile and link cleanly with no errors or warnings. If you have an error or warning, go back and look at the code provided and make sure that you have typed everything correctly.

I am a big proponent of compiling early and often. Typically, after every method that I write or any particularly tricky bit of code, I attempt to build. This is a good habit to get into, because it is much easier to narrow down compile errors if the amount of new code that you have added since your last successful compile is small. This practice also limits the number of errors or warnings that you receive. If you wait until you have written 2,000 lines before attempting to compile, you are likely to find the number of errors (or at least warnings) that you receive overwhelming. It is also sometimes difficult to track down the source of these errors because compiler and linker errors tend to be a little cryptic.

Your completed data model class should look like this:

```
#import "DataModel.h"
NSArray* myData;

@implementation DataModel

-(id)init
{
    if (self = [super init])
    {
        // Initialization code
        myData = [[NSArray alloc] initWithObjects:@"Albert", @"Bill", @"Chuck",
                  @"Dave", @"Ethan", @"Franny", @"George", @"Holly", @"Inez",
                  nil];
    }
    return self;
}

-(NSString*) getNameAtIndex:(int) index
{
    return (NSString*)[myData objectAtIndex:index];
}

-(int) getRowCount
{
    return [myData count];
}



@end
```

Displaying the Data

Now that you have the view and model in place, you have to hook them up using the controller. For a `table view` to display data, it needs to know what the data is and how to display it. To do this, a `UITableView` object must have a delegate and a datasource. The datasource coordinates the data from your model with the `table view`. The delegate controls the appearance and behavior of the `table view`. To guarantee that you have properly implemented the delegate, it must implement the `UITableViewDelegate` protocol. Likewise, the datasource must implement the `UITableViewDataSource` protocol.

Protocols

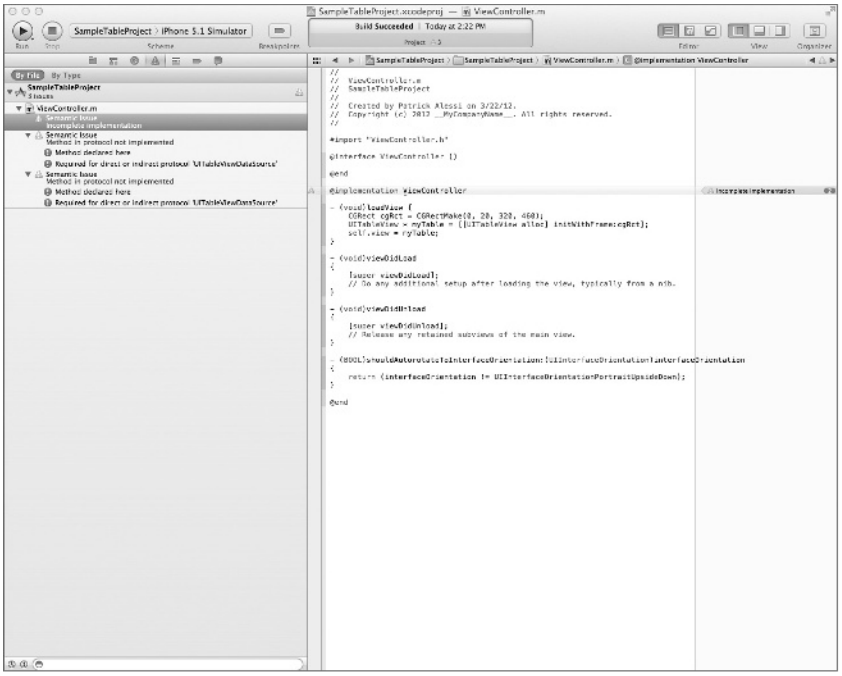
If you are familiar with Java or C++, protocols should also be familiar. Java interfaces and C++ pure virtual classes are the same as protocols. A *protocol* is just a formal contract between a caller and an implementer. The protocol definition states what methods a class that implements the protocol must implement. The protocol can also include optional methods.

Saying that a table view's delegate must implement the `UITableViewDelegate` protocol means you agree to a contract. That contract states that you will implement the required methods specified in the `UITableViewDelegate` protocol. Similarly, a class that will be set as the datasource for a table view must implement the required methods specified in the `UITableViewDataSource` protocol. This may sound confusing, but it will become clearer as you continue to work through the example.

To keep this example as simple as possible and to avoid introducing more classes, you make the `ViewController` the delegate and datasource for the table view. To do this, you have to implement the `UITableViewDelegate` and `UITableViewDataSource` protocols in the `ViewController`. You need to declare that the `ViewController` class implements these protocols in the header file. Change the `@interface` line in the `ViewController.h` header file to add the protocols that you plan to implement in angle brackets after the interface name and inheritance hierarchy like so:

```
@interface ViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate>
```

If you try to build your project now, you will get some warnings. Go to the Issue Navigator in the navigator pane by clicking the Issue Navigator icon at the top of the navigator pane or by using the shortcut Cmd+4. Your screen should look something like Figure 1-5.


SampleTableProject.xcodeproj — ViewController.m
Run
Stop
SampleTableProject › iPhone 5.1 Simulator
Scheme
Breakpoints
Build Succeeded | Today at 2:22 PM
Project ⚠ 3
Editor
View
Organizer
SampleTableProject › SampleTableProject › ViewController.m › @implementation ViewController
By File
By Type
Incomplete implementation

```
SampleTableProject
3 issues
ViewController.m
  Semantic Issue
  Incomplete implementation
  Semantic Issue
  Method in protocol not implemented
    Method declared here
    Required for direct or indirect protocol 'UITableViewDataSource'
  Semantic Issue
  Method in protocol not implemented
    Method declared here
    Required for direct or indirect protocol 'UITableViewDataSource'
```

```
//
//  ViewController.m
//  SampleTableProject
//
//  Created by Patrick Alessi on 3/22/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

- (void)loadView {
    CGRect cgRct = CGRectMake(0, 20, 320, 460);
    UITableView * myTable = [[UITableView alloc] initWithFrame:cgRct];
    self.view = myTable;
}

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (void)viewDidUnload
{
    [super viewDidUnload];
    // Release any retained subviews of the main view.
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}

@end
```

FIGURE 1-5: Using the Issue Navigator

You should see warnings associated with the compilation of `ViewController.m`; specifically, you should see the warnings "Semantic Issue incomplete implementation" and "Semantic Issue Method in protocol not implemented."

These warnings are clear. You have not implemented the protocols that you claimed you would implement. In fact, if you expand the issues in the Issue Navigator, you can use the navigator to see the required methods that you failed to implement. Expand the first "Semantic Issue Method in protocol not implemented" issue, and you will see two items. Click on the item labeled "Method declared here," and you will see the protocol definition in the editor pane with the method that you did not implement highlighted. In this case, it is the `tableView:numberOfRowsInSection:` method. If you click on the item labeled "Required for direct or indirect protocol 'UITableViewDataSource,'" the editor window changes back to your source code where you declared that you were going to implement the `UITableViewDataSource` protocol.

If you have any doubt about which methods are required to implement a protocol, a quick build will tell you and show you the exact method or methods that you have failed to implement.

Implementing the UITableViewDataSource Protocol

You can get rid of those warnings and move one step closer to a working application by implementing the `UITableViewDataSource` protocol.

Because you will be using the `DataModel` class in the `ViewController` class, you have to import the `DataModel.h` header file. In the `ViewController.h` header file, add the following `#import` statement just below the `#import <UIKit/UIKit.h>` statement:

```
#import "DataModel.h"
```

Now that you've imported the `DataModel` class, you have to create an instance variable of the `DataModel` type. In the `ViewController.m` implementation, add the following declaration below the `@implementation` keyword:

```
DataModel* model;
```

To actually create the instance of the model class, add the following code to the beginning of the `loadView` method:

```
model = [[DataModel alloc] init];
```

Now that you have an initialized model ready to go, you can implement the required `UITableViewDataSource` protocol methods. You can see from the compiler warnings that the methods you need to implement are `cellForRowAtIndexPath` and `numberOfRowsInSection`.

The `numberOfRowsInSection` method tells the table view how many rows to display in the current section. You can divide a table view into multiple sections. In the Contacts application, a letter of the alphabet precedes each section. In this example, you have only one section, but in Chapter 3, you see how to implement multiple sections.

To implement `numberOfRowsInSection`, get the number of rows that the datasource contains by calling the model's `getRowCount` method:

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section{
    return [model getRowCount];
}
```

If you look at the Issue Navigator now, you will see that the warning about not implementing `numberOfRowsInSection` is gone.

The `cellForRowAtIndexPath` method returns the actual `UITableViewCell` object that will display your data in the table view. The table view calls this method any time it needs to display a cell. The `NSIndexPath` parameter identifies the desired cell. So, what you need to do is write a method that returns the correct `UITableViewCell` based on the row that the table view asks for. You do that like so:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] ;
    }

    NSUInteger row = [indexPath row];
    cell.textLabel.text = [model getNameAtIndex:row];
    return cell;
}
```

The first few lines of code return a valid `UITableViewCell` object. I will not go into the details of exactly what is happening here because I cover it in detail in Chapter 3, which is dedicated to the `UITableView`. For now, suffice it to say that for performance purposes you want to reuse table view cells whenever possible, and this code does just that.

The last few lines of code find the row that the caller is interested in, look up the data for that row from the model, set the text of the cell to the name in the model, and return the `UITableViewCell`.

That's all there is to it. You should now be able to successfully build the project with no errors or warnings.

Delegates

In designing the iOS SDK and the Cocoa libraries in general, Apple engineers frequently implemented common design patterns. You've already seen how to use the MVC pattern in an application design. Another pattern that you will see all across the Cocoa and Cocoa touch frameworks is delegation.

In the *delegate* pattern, an object appears to do some bit of work; however, it can delegate that work to another class. For example, if your boss asks you to do some work and you hand it off to someone else to do, your boss doesn't care that you or someone else did the work, as long as the work is completed.

While working with the iOS SDK, you will encounter many instances of delegation, and the table view is one such instance. A delegate for the table view implements the `UITableViewDelegate` protocol. This protocol provides methods that manage the selection of rows, control adding and deleting cells, and control configuration of section headings along with various other operations that control the display of your data.

Finishing Up

The only thing left to do with the sample is to set the `UITableView`'s `delegate` and `DataSource` properties. Because you have implemented the `delegate` and `DataSource` protocols, in the `ViewController`, you set both of these properties to `self`.

In the `loadView` method of the `ViewController.m` file, add the following code to configure the datasource and the delegate for the table view:

```
[myTable setDelegate:self];
[myTable setDataSource:self];
```

The final code for the `ViewController.m` should look something like Listing 1-1.

LISTING 1-1: ViewController.m

```
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController
DataModel* model;

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section{
    return [model getRowCount];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
```

```
                    reuseIdentifier:CellIdentifier] ;
        }

        NSUInteger row = [indexPath row];
        cell.textLabel.text = [model getNameAtIndex:row];
        return cell;
    }

    - (void)loadView {
        model = [[DataModel alloc] init];

        CGRect cgRct = CGRectMake(0, 20, 320, 460);
        UITableView * myTable = [[UITableView alloc] initWithFrame:cgRct];
        self.view = myTable;

        [myTable setDelegate:self];
        [myTable setDataSource:self];

    }

    - (void)viewDidLoad
    {
        [super viewDidLoad];
        // Do any additional setup after loading the view, typically from a nib.
    }

    - (void)viewDidUnload
    {
        [super viewDidUnload];
        // Release any retained subviews of the main view.
    }

    - (BOOL)shouldAutorotateToInterfaceOrientation:
                        (UIInterfaceOrientation)interfaceOrientation
    {
        return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
    }

    @end
```

You should now be able to build and run your application in the simulator. You should see the table populated with the names that are contained in your `DataModel` class, as in Figure 1-6.

Congratulations! You have successfully built your first data-driven application! If you feel adventurous, feel free to go back and modify the `DataModel` to use a different datasource, like a text file.

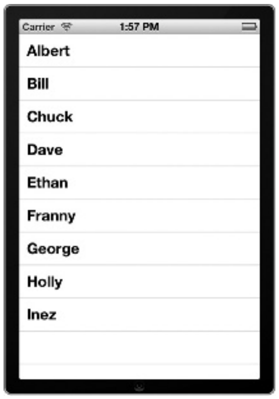


FIGURE 1-6: Running table view with data

FURTHER EXPLORATION

In this chapter, you learned how to build an iOS application that uses the `UITableView` control to display data. You also learned a little bit about design patterns — specifically the Model-View-Controller pattern that is prevalent in iOS application development. In the next chapter, you learn how to use the SQLite database as your datasource. Then, in Chapter 3, you master the `UITableView` control. By the end of Chapter 3, you should be able to build a data-centric iOS application on your own.

Design Patterns

If you are interested in writing maintainable, high-quality software, I highly recommend *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994). This is the bible of OO design patterns. The book illustrates each pattern with a UML model, very readable explanations of the patterns and their implementation, and code samples in both C++ and Smalltalk. If you don't already have this masterpiece of computer science, get it now — you won't regret it.

I would also recommend *Object-Oriented Design and Patterns* by Cay S. Horstmann (Wiley, 2005). Although the code in the book is in Java, you will find that the explanations of the patterns introduced are outstanding and will help you further understand the patterns and their importance in implementing high-quality software.

Even if you are not interested in either of these titles, do yourself a favor and search Google for "design patterns." There is a reason why there is a lot of literature on design patterns. It doesn't make any sense to try to reinvent the wheel. Others have already discovered the solutions to many of the problems that you will find in your software designs. These solutions have become design patterns. The point of these design patterns is to present well-tested designs to developers in a form that everyone can understand. The patterns are time proven and offer a common vocabulary that is useful when communicating your design to other developers.

Reading a Text File

If you are interested in making your simple table-viewing application more interesting, it is easy to read data from a text file. The following code snippet shows you how:

```
NSError *error;

NSString *textFileContents = [NSString
    stringWithContentsOfFile:[[NSBundle mainBundle]
    pathForResource:@"myTextFile"
    ofType:@"txt"]
    encoding:NSUTF8StringEncoding
    error:&error];

// If there are no results, something went wrong
if (fileContents == nil) {
    // an error occurred
    NSLog(@"Error reading text file. %@", [error localizedFailureReason]);
    }

    NSArray *lines = [textFileContents componentsSeparatedByString:@"\n"];
    NSLog(@"Number of lines in the file:%d", [lines count]  );
```

This code reads the contents of the file `myTextFile.txt`, which should be included in your code bundle. Simply create a text file with this name and add it to your Xcode project.

The first line declares an error object that will be returned to you should anything go wrong while trying to read your text file. The next line loads the entire contents of your file into a string.

The next line is an error handler. If you get `nil` back from the call to `stringWithContentesOfFile`, something went wrong. The error is output to the console using the `NSLog` function.

The next line breaks up the large string into an array separated by `\n`, which is the return character. You create an element in the lines array for each line in your file.

The final line outputs the number of lines read in from the file.

MOVING FORWARD

In this chapter, you learned how to build a simple data-driven application using an `NSArray` as your datasource. You also explored the project options available when creating a project in Xcode. Then you learned about the Model-View-Controller architecture and how the table view fits in with that design. Finally, you looked at the important concepts of protocols and delegates.

In the next chapter, you will learn how to get data from a more robust datasource, the SQLite database. This is the embedded database that is included as part of the iOS SDK. Learning to use this database will enable you to build rich, data-driven applications for iOS.