Chapter 5
Tables and Navigation

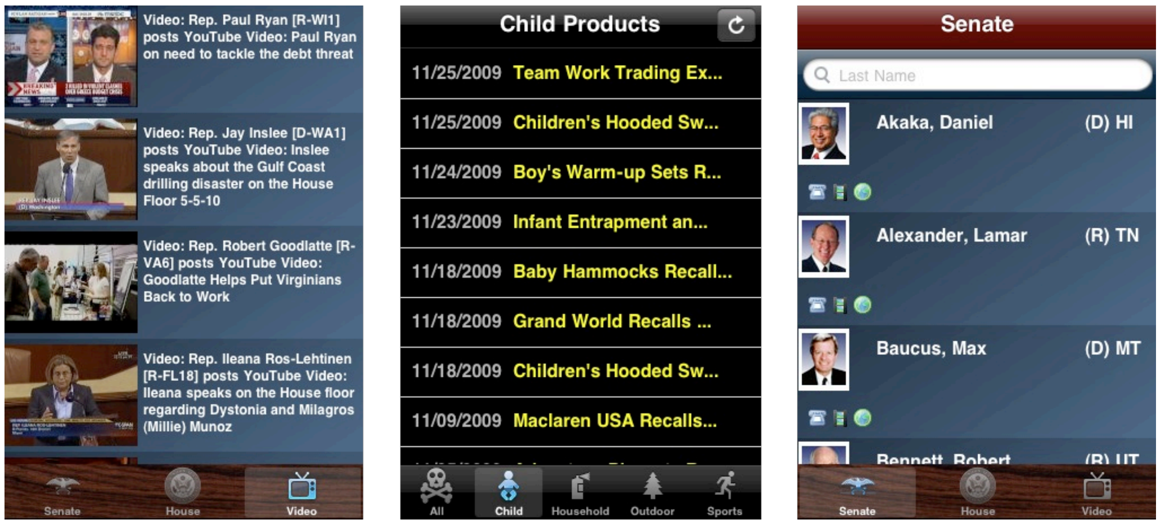
Tables are a fundamental user interface element in the iPhone SDK that are used to present lists of data. You will find tables used in many of the applications available in the app store as well as in several of the applications from Apple that ship with the device. In this chapter we will look at the how to create table driven applications and how they apply to navigation based scenarios. We will also introduce the open-source MonoTouch.Dialogs project and how it simplifies the creation of such applications.

Introduction to UITableView and UITableViewController

The UITableView and UITableViewController are the fundamental classes used to create table-drive user interfaces. This section introduces you to the concepts behind them, showing how to get an application up and running using tables.

What are tables used for?

Tables are used to present lists of data in an application. They are implemented via the UITableView class from UIKit. Short of gaming scenarios that typically take over the entire screen in a custom manner, you will find tat most applications use tables. In fact many applications are built entirely on tables. UITableView comes with several stock design features that you can take advantage of. It is also quite customizable, allowing you to control the look and feel of your application even further if the stock support isn't adequate for your purposes. Figure 5.1 shows examples of some applications using UITableViews.



Video: Rep. Paul Ryan [R-WI1] posts YouTube Video: Paul Ryan on need to tackle the debt threat

Video: Rep. Jay Inslee [D-WA1] posts YouTube Video: Inslee speaks about the Gulf Coast drilling disaster on the House Floor 5-5-10

Video: Rep. Robert Goodlatte [R-VA6] posts YouTube Video: Goodlatte Helps Put Virginians Back to Work

Video: Rep. Ileana Ros-Lehtinen [R-FL18] posts YouTube Video: Ileana speaks on the House floor regarding Dystonia and Milagros (Millie) Munoz

11/25/2009 Team Work Trading Ex...
11/25/2009 Children's Hooded Sw...
11/24/2009 Boy's Warm-up Sets R...
11/23/2009 Infant Entrapment an...
11/18/2009 Baby Hammocks Recall...
11/18/2009 Grand World Recalls ...
11/18/2009 Children's Hooded Sw...
11/09/2009 Maclaren USA Recalls...

Akaka, Daniel (D) HI
Alexander, Lamar (R) TN
Baucus, Max (D) MT
Bennett, Robert (R) UT

***insert figure 5.1 05fig01.tif

Figure 5.1 Applications Using UITableViews

The UITableView class is responsible for the presentation of data. Being a subclass of UIView, it is the view in the MVC pattern. The UITableView delegates responsibility to a data source for populating from a model. In Objective-C the data source is any class that conforms to the UITableViewDataSource protocol. To respond to behavior that occurs in a UITableView, such as one of the table's rows being selected, the UITableView sends messages to an implementation of the UITableViewDelegate protocol. For the controller UIKit contains a class called UITableViewController that contains a UITableView set as it's view with the aforementioned two protocols already set to the UITableViewController itself. You'll recall Objective-C protocols are similar to C# interfaces except protocols allow optional methods.

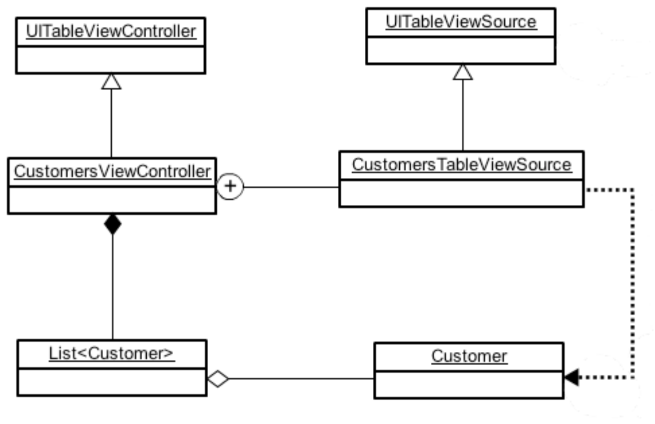
Note

You can also use a UIViewController directly as the controller for a UITableView and set up the table, delegate and data source manually. I'll show you an example of that later in the chapter.

Lacking a mechanism in C# interfaces for the optional method support afforded by Objective-C protocols, in MonoTouch Objective-C protocols are defined in classes. Therefore, in MonoTouch you cannot get the default behavior of the UITableViewController being both the delegate and the data source for its UITableView directly. The common design pattern in MonoTouch is to create nested classes for the data source and the delegate within the UITableViewController subclass. For convenience MonoTouch wraps the data source and delegate together into a single class called TableSource so that you only need to create nested class in your UITableViewController. Let's take a look at an example to help illustrate these concepts and introduce some of the stock capability of UITable and its related classes.

Displaying Data in a UITableView

We are going to create an application that presents a customer. A customer has a first name, last name and optionally a note for our purposes here. We will enhance this class as we proceed through the chapter. The class diagram for the application is shown in Figure 5.2



***insert figure 5.2 05fig02.tif

Figure 5.2 Initial Class Diagram For Customer Viewer

Start by creating a new MonoTouch iPhone Window-based Project called LMT5-1. Add a new class to the project, which we will use to create our UITableViewController, naming it CustomersViewController. Make the class derive from UITableViewController.

```
using System;
using MonoTouch.UIKit;

namespace LMT51
{
    public class CustomersViewController : UITableViewController
    {
        public CustomersViewController ()
        {
        }
    }
}
```

Deriving from UITableViewController gives us a UIViewContoller that has a UITableView for its view. In Objective-C this class would also be pre-configured as the delegate and datasource for the UITableView. In MonoTouch we could either create two nested classes and set the Delegate and Datasource properties on the TableView or use the combined UITableView. Let's do the latter, creating a nested class called CustomersTableViewSource, subclassing UITableViewSource.

```
…
class CustomersTableViewSource : UITableViewSource
{
}
```

Now we need to set the UITableView, accessed via the UITableViewController's TableView property, to point to an instance of this class. You might be inclined to create it inside the CustomerViewController's constructor. This is not a good practice in general for a couple reasons. First, views in CocoaTouch are created on demand. In a multi-view application, this means that you wouldn't incur the overhead of creating a view until it is needed for presentation. Second, just because a view controller may be kept in memory, doesn't mean its view should be. When a view is not displayed its memory can be reclaimed. When memory gets low, view controllers whose views are not currently being displayed will be release their views. When the view are needed again for display, ViewDidLoad will be called If you implemented code against the view in the constructor of the view controller, your code would not get executed again. Therefore, ViewDidLoad is a good place to do any initialization of the UITableView that is required one time for the view's instance.

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    TableView.Source = new CustomersViewController ();
}
```

The CustomersViewController will keep a reference to the data that the CustomersTableviewSource will tell the table to display (via the UITableView's abstraction of the binding to the Objective-C UITableViewDataSource protocol). At a minimum the number of rows to present and the data to present in each row are returned.

Note

MonoTouch makes the methods that are required in the Objective-C protocol abstract in the UITableViewDataSource so that you won't be able to leave them out inadvertently.

For the number of rows you simply need to implement the RowsInSection method. This will be used to create the proper number of rows. To get the object to put in each row, represented by a UITableViewCell, we need to implement GetCell as well. The basic steps are:

1. Create a model class
2. Add a collection of the model to the UITableViewController's subclass
3. Pass the UITableViewController to the data source
4. Implement RowsInSection to return the number of items in the collection
5. Implement GetCell to return a UITableCell for the data encapsulated by each item in the collection

For the model we need a Customer class. Create a new class named Customer containing the code in Listing 5.2.

```
using System;

namespace LMT51
{
    public class Customer
    {
        string _fName;
        string _lName;

        public string FName {
            get { return this._fName; }
            set { _fName = value; }
        }

        public string LName {
            get { return this._lName; }
            set { _lName = value; }
        }

        public Customer (string fName, string lName)
        {
            _fName = fName;
            _lName = lName;
        }

        public string Note { get; set; }
    }
}
```

Listing 5.2 Model Class Representing a Customer

Next add a List<Customer> to the CustomersViewController. This list will be filled with customers when we create the CustomersViewController. Since it is required we'll add it to the constructor as well.

```
public class CustomersViewController : UITableViewController
{
    List<Customer> Customers { get; set; }

    public CustomersViewController (List<Customer> customers)
    {
        Customers = customers;
    }
…
```

In the CustomersTableViewSource we pass a reference to the view controller since we'll need to access the Customers to get the count for the number of rows and each customer to present in a UITableViewCell.

```
class CustomersTableViewSource : UITableViewSource
{
    CustomersViewController _vc;

    public CustomersTableViewSource (CustomersViewController vc)
    {
        _vc = vc;
    }
…
```

This implementation of RowsInSection need only return the number of Customers in the list. Ignore the section parameter for now. We'll discuss that momentarily.

```
public override int RowsInSection (UITableView tableview, int section)
{
    return _vc.Customers.Count;
}
```

The GetCell method will be called once for each row in the table as the rows appear on the screen. What we want do here is use the index information, passed in via the NSIndexPath, to pull out the proper Customer object and fill a UITableViewCell with its data. An NSIndexPath contains two pieces of information, the row and the section. A UITableView can have multiple sections, each presenting its own list of data. The UITableView asks its DataSource for each row in each section. Since we only have one section, we can concern ourselves with just the row.

Note
Later in the chapter we will extend this example to include multiple sections.

To get the UITableView cell for the row (in a given section if there were more than one) that that UITableView requested we need to:

1. Get the row from the indexPath that is passed in.
2. Get the customer that corresponds to this row.
3. Get or create a UITableView cell.
4. Populate the cell with data the customer's data (or whatever model object you have).
5. Return the cell.

In order to get a cell we either reuse a previously created cell or create a new one. In order to conserve resources, UITableViews add cells that have been created to a cache when the cells move off the display as you scroll the table. This way cells can be reused. As the user scrolls the table, it will ask the DataSource for cells that need to be displayed. In the GetCell implementation you simply ask for a cell from the cache of cells that are no longer visible and populate it with the new data, only creating a new cell if one isn't available. As tables can contain many types of cells potentially, either to present different data or to change the presentation in some cases, you need to create cells with an identifier indicating what type of cell they are, so that you can retrieve the proper type of reusable cell as needed when the user scrolls. In our example we currently only have one type of cell to present the customer's data, so that is the only identifier well need.

Listing 5.3 shows CustomersTableViewSource containing a GetCell implementation to either get a previously created cell or create a new one, populating it with the proper customer data in either case.

```
class CustomersTableViewSource : UITableViewSource
{
    CustomersViewController _vc;

    const string CUSTOMER_CELL = "customerCell";

    public CustomersTableViewSource (CustomersViewController vc)
    {
        _vc = vc;
    }

    public override int RowsInSection (UITableView tableview,
        int section)
    {
        return _vc.Customers.Count;
    }

    public override UITableViewCell GetCell (UITableView tableView,
        NSIndexPath indexPath)
    {
        int row = indexPath.Row;

        UITableViewCell cell =
            tableView.DequeueReusableCell (CUSTOMER_CELL);

        if (cell == null)
           cell = new UITableViewCell
```

```
                (UITableViewCellStyle.Default, CUSTOMER_CELL);

            Customer aCustomer = _vc.Customers[row];

            cell.TextLabel.Text = String.Format ("{0} {1}",
                aCustomer.FName, aCustomer.LName);

            return cell;
        }
    }
```

Listing 5.3 CustomersTableViewSource Implementing GetCell

Notice the second argument to the UITableViewCell constructor. This is where we pass in a string constant identifying the cell as a customer cell, so that we can later retrieve it for reuse using the UITableView's DequeueReuseableCell method.

To make use of our table, we need to create some customers and add them to the controller. Normally you would make a call for some external store such as a local storage or via web service. In this example we'll just create a few customers in code. Add the following code to the AppDelegate's FinishedLaunching method to create the CustomersViewController with some sample Customer objects:

```
CustomersViewController _customersVC;

public override bool FinishedLaunching (UIApplication app,
    NSDictionary options)
{
    _customersVC = new CustomersViewController (new List<Customer> {
        new Customer ("Jane", "Doe"),
        new Customer ("Joe", "Smith") });

        window.AddSubview (_customersVC.View);

        window.MakeKeyAndVisible ();

        return true;
}
```

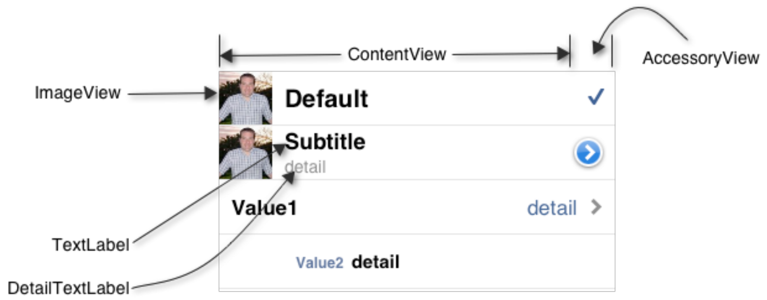
Note

Keep a reference to the CustomersViewController so that it doesn't get garbage collected.

If you run the application now you will see the table we created populated with the names of the customers. The look of the cells is dictated by the cell style that we request when creating the cell. In Listing 5.3 note the first argument to the UITableViewCell constructor where we indicated what style of cell we wanted. The stock UITableViewCell class from Apple supports four different styles, (although you can also subclass it to customize the style to your liking, which we will do later in the chapter). Let's discuss the parts of a UITableViewCell and how we can control the look using the UITableViewCellStyle.

UITableViewCell Parts and Styles

UITableViewCell's consist of two views, a ContentView and an AccessoryView. The AccessoryView is used to display an image that indicates how the row can be acted upon, such as a checkmark for selection, a detail disclosure to indicate a master-detail scenario, or a detail button to indicated further details of the object the row represents additional to what is available in a master-detail navigation scenario. The ContentView is made up of three subviews, a TextLabel, a DetailTextLabel and optionally an ImageView, with the layout and display of the ContentView controlled by the UITableViewCellStyle enumeration (Figure 5.3).



***insert figure 5.3 05fig03.tif

Figure 5.3 UITableViewCell Parts and Styles

Let's go back to our example and change the cells to demonstrate showing a different style in some cases, allowing us to further illustrate cell reuse in light of varying cell styles. We are going to add support for optionally showing some additional text about customers below their names. We'll use the Note property of the Customer to hold this data.

Change the code in the AppDelegate, where we created our Customer objects, to populate the note in some cases. For example,

```
public override bool FinishedLaunching (UIApplication app,
                                        NSDictionary options)
{
    _customersVC = new CustomersViewController (new List<Customer> {
        new Customer ("Jane", "Doe"),
        new Customer ("Joe", "Smith"),
        new Customer ("Steve", "Jones") {Note = "Send email" },
        new Customer ("Alice", "Smith") {Note = "New customer"}
});
…
```

Now what we want to do is use a cell with a style of UITableViewCellStyleSubtitle for the customers with a note while keeping a UITableViewCellStyleDefault for those without a note. In the new GetCell implementation we check to see if the note is present and use that to switch between cell styles and subsequently to populate the DetailTextLabel.Text with the note as shown in Listing 5.4.

```
const string CUSTOMER_CELL = "customerCell";
const string CUSTOMER_CELL_WITH_NOTE = "customerCellWithNote";

…
```

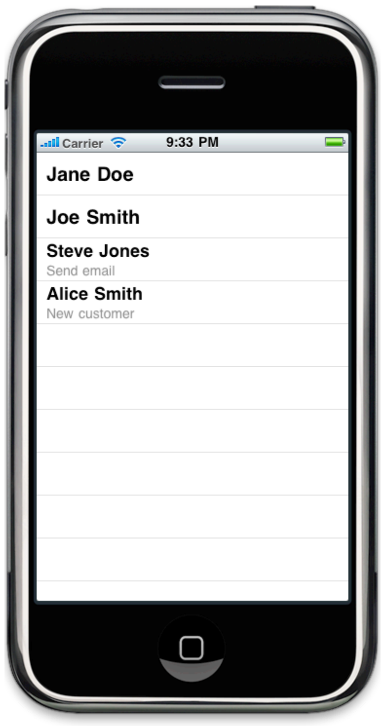
```
public override UITableViewCell GetCell (UITableView tableView,
    NSIndexPath indexPath)
{
    UITableViewCell cell;

    int row = indexPath.Row;

    Customer aCustomer = _vc.Customers[row];

    if (String.IsNullOrEmpty (aCustomer.Note)) {
        cell = DequeueOrCreateCell (tableView,
            UITableViewCellStyle.Default, CUSTOMER_CELL);
    } else {
        cell = DequeueOrCreateCell (tableView,
            UITableViewCellStyle.Subtitle, CUSTOMER_CELL_WITH_NOTE);
        cell.DetailTextLabel.Text = aCustomer.Note;
    }

    cell.TextLabel.Text = String.Format ("{0} {1}", aCustomer.FName,
        aCustomer.LName);

    return cell;
}

UITableViewCell DequeueOrCreateCell (UITableView tableView,
    UITableViewCellStyle cellStyle, string cellIdentifier)
{
    UITableViewCell cell;
    cell = tableView.DequeueReusableCell (cellIdentifier);
    if (cell == null)
        cell = new UITableViewCell (cellStyle, cellIdentifier);
    return cell;
}
```

Listing 5.4 GetCell Implementation With Different Cells Styles

Running the app, you will see that the two different style cells are shown within the same table (Figure 5.4).

Note

The same approach used to distinguish between differently styled cells can also be used to distinguish different cell types that are subclasses of UITableViewCell.



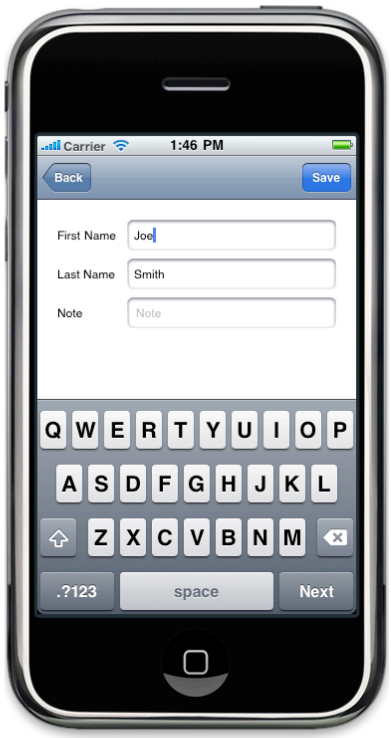
***insert figure 5.4 05fig04.tif

Figure 5.4 UITableViewShowing Multiple Cell Styles

Using Tables and Navigation

Tables lend themselves nicely to navigating hierarchies of data. Through a combination of the UITableViewDelegate protocol, (which recall is abstracted by MonoTouch in the UITableViewSource class), and UINavigationController, you can add this capability to any application. Also, detail views are a common way to edit the content of the data presented in a table cell.

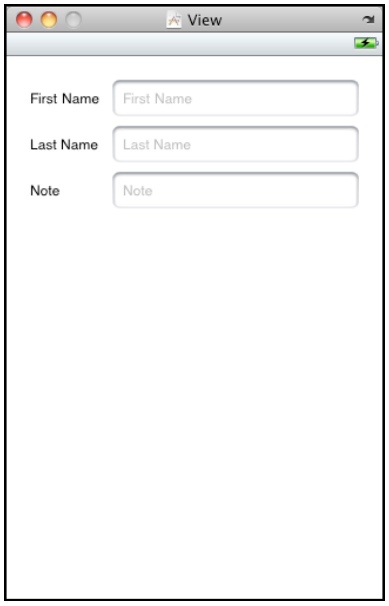
Let's add support to our customer app to show a second view to edit the content of a customer, including the first name, last name and note, as shown in Figure 5.5.



***insert figure 5.5 05fig05.tif

Figure 5.5 Customer App With Detail Editing View

The first thing we need to add is a new view and controller for the detail view. Add a new iPhone View with Controller to the project called CustomerDetailViewController. Add three UITextFields in Interface Builder for the first name, last name and note, connecting outlets for each to the File’s Owner. Also, while in Interface Builder, set the Return Key to “Next” for each UITextField. The view in IB should look something like Figure 5.6



***insert figure 5.6 05fig06.tif

Figure 5.6 View in CustomerDetailsViewController.xib

We want to have the name UITextField get focus when the view is loaded, presenting the keyboard for it. Also, we want to be able to move between the text fields by selecting “Next” for the keyboard’s return key. Add the following code to CustomerDetailViewController’s ViewDidLoad method to accomplish this (Listing 5.4).

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();

    firstNameTextField.BecomeFirstResponder ();

    firstNameTextField.ShouldReturn += (tf) =>
    {
        lastNameTextField.BecomeFirstResponder ();
        return true;
    };

    lastNameTextField.ShouldReturn += (tf) =>
    {
        noteTextField.BecomeFirstResponder ();
        return true;
    };

    noteTextField.ShouldReturn += (tf) =>
    {
        firstNameTextField.BecomeFirstResponder ();
        return true;
    };
}
```

Listing 5.4 FirstResponder Code in CustomerDetailViewController

The CustomerDetailViewController will be managing the view for a single customer, so it will have an instance of a Customer as well. Let’s include the customer in the constructor for the controller.

```
public partial class CustomerDetailViewController : UIViewController
{
    Customer _customer;
    ...
    public CustomerDetailViewController (Customer c) :
        base("CustomerDetailViewController", null)
    {
        Initialize ();
        _customer = c;
    }
…
```

Our goal is to create the CustomerDetailViewController, populate it with a Customer instance and show its view when a row is selected in the CustomerViewController's table view. We will use the UINaviagtionController class to accomplish this.

The UINavigationController is a class that manages a stack of UIViewControllers that are designed to present various views of related data. The UINavigationController itself is also a UIViewController and as such has a view. As UIViewControllers are pushed onto the stack or popped from the stack, the view of the controller at the top of the stack is shown in a subview of the UINavigationController's view along with another subview, containing a UINavigationBar, which allows the user to navigate the controller stack. See Figure 5.7.

TODO: figure 5.7

***insert figure 5.7 05fig07.tif

Figure 5.7 Composition of a UINavigationController

When you first create a UINavigationController, nothing has been added to its stack, so you have to add a view controller to it. This first view controller is the UINavigationController's RootViewController and must always be present. In our case we want the RootViewController to be the CustomersViewController. Modify the AppDelegate's FinshedLaunching method, where we originally created the CustomersViewController instance, to wrap the CustomersViewController in a UINavigationController, adding the UINavigationController's view to the window (Listing 5.5).

```
public partial class AppDelegate : UIApplicationDelegate
{
    UINavigationController _navController;
    CustomersViewController _customersVC;

    public override bool FinishedLaunching (UIApplication app,
        NSDictionary options)
    {
        _customersVC = new CustomersViewController (
            new List<Customer> {
                new Customer ("Jane", "Doe"),
                new Customer ("Joe", "Smith"),
                new Customer ("Steve", "Jones"){
                    Note = "Send email"},
                new Customer ("Alice", "Smith"){
                    Note = "New customer"}
            });

            _navController = new UINavigationController (_customersVC);

            window.AddSubview (_navController.View);

            window.MakeKeyAndVisible ();

            return true;
        }
}
```

Listing 5.5 Adding CusotmersViewController to a UINavigationController

If you run this you will have the CustomersViewController's view presented as before but this time it will be the subview of a UINavigationController's view The blue bar at the top is the UINavigationBar, also a subview of the UINavigationController's view. We'll get to how the UINavigationBar is populated with content shortly but first let take advantage of our new navigation controller to present the CustomerDetailViewController's view when a customer is selected.

In the CustomersTableViewSource implement RowSelected, which will be called by the table view when one of its rows is selected. Within the RowSelected method we can create CustomerDetailViewController and push it onto the navigation controller's stack.

```
class CustomersTableViewSource : UITableViewSource
{
    CustomersViewController _vc;
    CustomerDetailViewController _customerDetail;
    …

    public override void RowSelected (UITableView tableView,
        NSIndexPath indexPath)
    {
        Customer selectedCustomer = _vc.Customers[indexPath.Row];

        _customerDetail =
            new CustomerDetailViewController (selectedCustomer);

        _vc.NavigationController.PushViewController (
            _customerDetail, true);
    }
…
}
```

Notice how we accessed the NavigationController property of the CustomersViewController. By participating in a UINavigationController, meaning it was added to its stack; the UIViewController's property will point to the UINavigationController instance it was added to. With the reference to the navigation controller in hand we can call PushViewController to add a new view controller, in this case the CustomerDetailViewController, to the top of the stack. When pushed onto the navigation controller's stack, the CustomerViewController's view will be presented in the subview of the UINavigationController's view. Setting the animated argument to true will cause it to slide into view horizontally. We want the CustomerDetailViewControllers's view to display the data for the Customer object that was passed in. Let's take care of that when the view is loaded.

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    …
    firstNameTextField.Text = _customer.FName;
    lastNameTextField.Text = _customer.LName;
    noteTextField.Text = _customer.Note;
}
```

Build and run the project now. When you select a row you, the CustomerDetailViewController's view will slide in and the various text field will be populated with the appropriate customer data. Also note the left side of UINavigationBar shows “Back” button when the CustomerDetailViewController’s view shown. Selecting it takes you back to CustomersViewController’s view as you would expect.

The UINavigationBar is a subview of the UINavigationController’s view as we mentioned earlier. The subviews of the UINavigationBar are not intended for direct access however. Instead, every UIViewController has a property called NavigationItem that is of type UINavigationItem. The NavigationItem’s purpose in life is to provide the ui that will make up the UINavigationBar. When a UINavigationBar is ready to be drawn, it looks into the NavigationItem for the controller at the top of the UINavigationController’s stack for three properties named TitleView, LeftBarButtonItem and RightBarButtonItem. Using these along the NavigationBar is able to present itself appropriately. For example add the following code to the bottom of CustomerDetailViewController.ViewDidLoad.

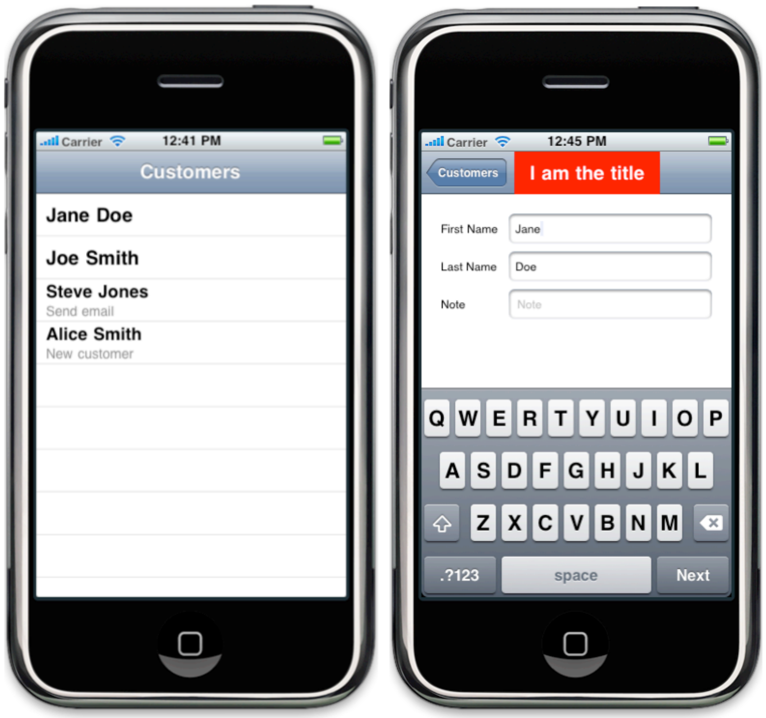
```
UILabel redLabel = new UILabel ();
redLabel.Frame = new System.Drawing.RectangleF (0, 0, 150, 44);
redLabel.TextAlignment = UITextAlignment.Center;
redLabel.Font = UIFont.BoldSystemFontOfSize (20);
redLabel.BackgroundColor = UIColor.Red;
redLabel.TextColor = UIColor.White;
redLabel.Text = "I am the title";
this.NavigationItem.TitleView = redLabel;
```

Now when you run the application and navigate to the customer detail, the NavigationBar will be populated with a hideous red label containing the string “I am the title.” The NavigationBar looked into the NaviagtionItem for the TitleView and populated itself accordingly.

If all you want to do is set the string for the title in the TitleView, you can simply set the Title property of the UIViewController. Additionally, setting the Title will fill the text of the LeftBarButtonItem with the title of the next controller down in the stack, rather than the string “Back.” For example in CustomersViewController.ViewDidLoad() add the following line to set the title.

```
this.Title = "Customers";
```

This causes the title to be set to “Customers” in the NavigationBar when the CustomersViewController’s view is displayed as well as in the LeftBarButtonItem when the CustomerDetailViewController’s view is displayed (Figure 5.8).



***insert figure 5.8 05fig08.tif

Figure 5.8 Result of Setting CustomersViewController.Title

Note

Actually, the LeftBarButtonItem and RightBarButtonItem are not views that get displayed, but instead are containers for the buttons that are rendered in the UINavigationBar, much like the NavigationItem itself. MonoTouch abstracts these nicely however, even providing a .Net style click event.

At this point we have a secondary view controller in place, the CustomerDetailViewController, to present a view for editing the content of a customer, but we haven't added the code to save the changes back to the customer object. For this example let's add a save button to the RightBarButtonItem of the NavigationBar to commit any changes and navigate back to the CustomersViewController, showing the updated customer data in the table.

Like everything else in the NavigationBar, we can add the save button via the NavigatonItem. In this case we will use the RightBarButtonItem. UIKit contains several stock buttons for use in UINavigationBars (and UIToolBars). MonoTouch makes these available through the UIBarButonSystemItem enumeration. Add code to the CustomerDetailViewController to create the save button.

```
UIBarButtonItem _saveButton;
…
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    _saveButton = new UIBarButtonItem (UIBarButtonSystemItem.Save);
    _saveButton.Clicked += Handle_saveButtonClicked;
    this.NavigationItem.RightBarButtonItem = _saveButton;
    …
}
```

When the save button is selected, we want to update the customer object and go back tot the previous screen, displaying the new data in the table. To update the customer we can simply fill its properties in Clicked event handler and then call the UINavigationController's PopViewControllerAnimated method to go back to the CustomersViewController.

```
void Handle_saveButtonClicked (object sender, EventArgs e)
{
    _customer.FName = firstNameTextField.Text;
    _customer.LName = lastNameTextField.Text;
    _customer.Note = noteTextField.Text;
    this.NavigationController.PopViewControllerAnimated (true);
}
```

Back in the CustomersViewController call TableView.ReloadData () when the view appears to force the table to show the newly updated customer data.

```
public override void ViewWillAppear (bool animated)
{
    base.ViewWillAppear (animated);
    TableView.ReloadData ();
}
```

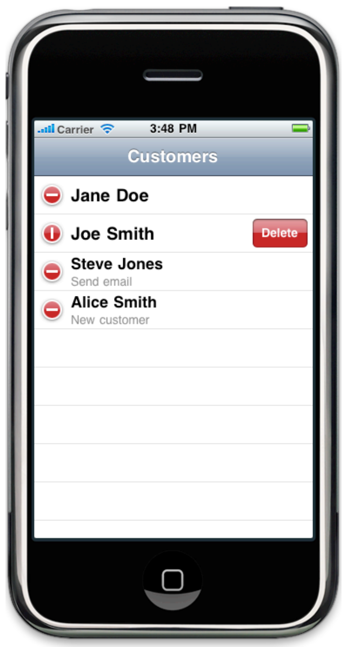
Note

This will cause the entire table to reload its data every time, so you may want to take measures to only reload the data when it has actually changed.

Run the app, select a customer and make some changes to the data. Selecting save will cause the changes to be committed to the customer object in memory and presented in the table.

UITableView's Built-In Editing Support

So far we have seen how to display data in a table and navigate to detail views to present and change the content of the data. The UITableView class also comes with support for adding, deleting and reordering cells and their associated data, including some nice, built-in user interface features. In order to take advantage of this functionality, we need to set the Editing property of the UITableView to true. Doing this causes the table to enter editing mode, which without any code, gives the standard delete interface, with a delete icon on the left of each row. Selecting any delete icon causes a delete button to animate in from the right of the row, as you have probably seen in several iPhone apps.



***insert figure 5.9 05fig09.tif

Figure 5.9 UITableView with Editing property set to true

Of course, nothing actually gets deleted as we haven't added any code to do that, but it's nice to get common ui functionality for free. In order to manipulate customer rows when editing, we need to implement some additional methods on the UITableViewSource. First let's add a button to the NavigationBar to allow us to edit the table's rows and then we'll add the necessary implementation to the CustomersTableViewSource.

We could create any UIBarButtonItem ourselves and set the NavigationItem's RightBarButtonItem, handling the clicked event to toggle the TableView.Editing between true and false. Since our controller is part of a UINavigationController, there's an even easier way. As this is such a common scenario, UIKit contains built in edit button support. Simply setting one of the NavigationItem's BarButtonItems to the controllers built in EditButtonItem results in a button capable of toggling the table's editing state without any additional code on your part.

```
public override void ViewDidLoad ()
{
     base.ViewDidLoad ();
     this.Title = "Customers";

     this.NavigationItem.RightBarButtonItem = this.EditButtonItem;
     …
```

Again, more ui functionality for free. We will have to write some code to actually make things change however. Let's take care of deleting a row and the underlying customer object. In order to make this happen we need to implement CommitEditingStyle on the UITableViewSource. Add and empty implementation of this to the CustomersTableViewSource and run the application.

```
public override void CommitEditingStyle (UITableView tableView,
UITableViewCellEditingStyle editingStyle, NSIndexPath indexPath)
{
}
```

Apply a swipe gesture to any row and a delete button will animate in from the left. The presence of the CommitEditingStyle was enough to cause the swipe gesture to come to life. Pretty cool, huh! Now, to make the delete take place we need to implement this method. Add the following code to handle deleting any rows from the table along with the underlying customer data.

```
public override void CommitEditingStyle (UITableView tableView,
UITableViewCellEditingStyle editingStyle, NSIndexPath indexPath)
{
    if(editingStyle == UITableViewCellEditingStyle.Delete){
        _vc.Customers.RemoveAt (indexPath.Row);
        tableView.DeleteRows (new NSIndexPath[]{indexPath},
            UITableViewRowAnimation.Middle);
    }
}
```

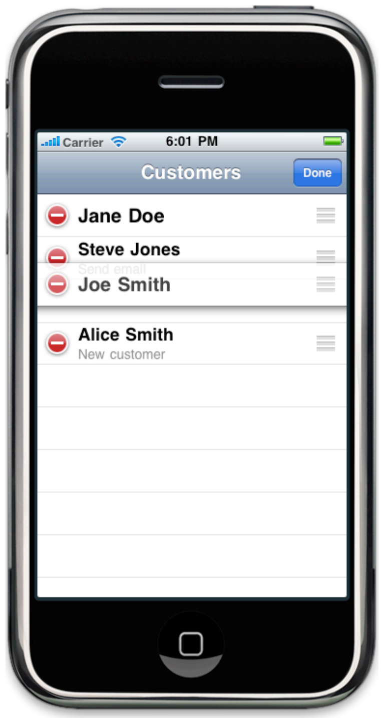
Deleting the rows is a three steps process:

1. Check if the edit operation requested was a delete.
2. Remove the row from the underlying data.
3. Delete the row from the table.

Similarly, you can also implement support for reordering the rows in a table. To enable this we needto implement another UITableViewSource method, MoveRow. As we did earlier for CommitEditingStyle, add an empty implementation of MoveRow to the CustomersTableViewSource class.

```
public override void MoveRow (UITableView tableView, NSIndexPath
sourceIndexPath, NSIndexPath destinationIndexPath)
{
}
```

With this in place, run the application, selecting the Edit button. Again, simply having this method in place enabled a move row icon to appear on the right of each row. If you select the icon and drag it a new position in the table, you will see the row dragged along with it. Releasing the row cause an animated drop and insert at the new position in the table (Figure 5.10).



***insert figure 5.10 05fig10.tif

Figure 5.10 Animated drag and drop reordering of table rows

The ui is entirely taken care of for us in this case; we only need to take care of moving the row appropriately in the backing data model, which we can do in the MoveRow implementation.

```
public override void MoveRow (UITableView tableView, NSIndexPath
sourceIndexPath, NSIndexPath destinationIndexPath)
{
    Customer c = _vc.Customers[sourceIndexPath.Row];
    _vc.Customers.RemoveAt (sourceIndexPath.Row);
    _vc.Customers.Insert (destinationIndexPath.Row, c);
}
```

Now when we move the row in the associated Customer object is moved in the backing list as well. To demonstrate this after reordering the rows, navigate to the detail view and then back for any row, which will result in our code reloading the table data. Notice that the order is maintained. If you this with the above implementation, the order will return to its original state.

Note

If you close the app and remove it from the background processing, the original list will reappear because we only changed things in memory. We will look at how to persist changes such as this later in the book.

We've looked at how to delete and reorder rows in a table. To complete the story let's see how to go about creating entirely new customer rows and add them to the table. In order to handle adding new rows we will take the approach of creating an "Add Customer" row to the table when in edit mode. UITableView's provide built in support for row that trigger an insert row action via the UITableViewEditingStyle. By creating a row with a UITableViewEditingStyle of Insert, you will get back a cell that contains a green plus icon signifying adding a row when the table is editing. The main difference in this case, as opposed to the delete and reorder cases, is the row in question will not be containing data but will live only for the purpose of triggering the addition of a new data row. Therefore some special handling will need to be added to "make room" for the additional add row and also to account for it if you are supporting row reordering. The basic steps to implement the add customer row are:

When editing,

1. Add a new row to the table for the "Add Row" functionality.
2. Increment the row count to accommodate the new row.
3. Set the UITableViewEditingStyle to Insert for this extra row.
4. Create the cell for the extra row, to whatever design you'd like.
5. If supporting row reordering, disable it for the extra row (since it isn't presenting data).
6. If supporting row reordering, don't allow a drop past the extra row (since that would be beyond the upper bound of you actual data collection, the customer list in this case).
7. Create a new data object (Customer), adding it to the backing store (the customer list) when the "Add Row" is selected.
8. Insert a new row to present the new data object.

When not editing, do things like you did before, adding rows only for the data contained in the customer list. The only additional step needed when not editing is to remove the extra "Add Row."

Note

We could also place an add row button elsewhere in the ui, such as in a toolbar, and navigate to a detail view to create the new customer's data, adding it to the list and reloading the table on return. This is often a more suitable experience (see the Notes application that ships with iPhone for example). For this exercise we're doing it inline specifically to present the insert editing style that ships with UIKit, although for your particular application's use case, use the external add button/view approach if it makes more sense.

For adding a new row to accommodate the "Add Row," we can override SetEditing on UITableViewController. You may be tempted to handle this in a clicked event handler for the edit button. This would prevent the table form ever entering editing however. SetEditing is called whenever the editing state changes, so we can also take care of removing the extra row here when editing ends.

```
public override void SetEditing (bool editing, bool animated)
{
    base.SetEditing (editing, animated);
    (TableView.Source as CustomersTableViewSource).IsEditing = editing;
    if (editing) {
        TableView.InsertRows (new NSIndexPath[] {
            NSIndexPath.FromRowSection (Customers.Count, 0) },
            UITableViewRowAnimation.None);
    } else {
        TableView.DeleteRows (new NSIndexPath[] {
            NSIndexPath.FromRowSection (Customers.Count, 0) },
            UITableViewRowAnimation.None);
    }
}
```

Notice we set our own IsEditing property on CustomersTableViewController. We use this property internally to prevent editing case of swipe to delete, which would happen without the extra row in place, from crashing our app duwe to the row count mismatch that would happen if we just checked on the table's editing state.

```
public bool IsEditing { get; set; }

public CustomersTableViewSource (CustomersViewController vc)
{
    IsEditing = false;
    _vc = vc;
}
```

Now that you have added a new row when editing, you need to make room for it by increasing the row count by one when editing.

```
public override int RowsInSection (UITableView tableview, int section)
{
    int c = _vc.Customers.Count;

    if (_vc.TableView.Editing && IsEditing) {
        c++;
    }
    return c;
}
```

These steps will get you an extra row, but you still need to create the proper editing style and subsequently the row itself. UITableViewSource has an EditingStyleForRow method you can override to set the UITableViewEditStyle for each row. You wan tall the rows to be a Delete style except for the extra "Add Row," which will bean Insert style.

```
public override UITableViewCellEditingStyle EditingStyleForRow
(UITableView tableView, NSIndexPath indexPath)
{
    UITableViewCellEditingStyle editingStyle;

    if (indexPath.Row < _vc.Customers.Count) {
        editingStyle = UITableViewCellEditingStyle.Delete;
    } else {
        editingStyle = UITableViewCellEditingStyle.Insert;
    }
    return editingStyle;
}
```

Create the special “Add Row” where all the other rows are created, in GetCell, only this time customizing the cell content specifically for the purpose of adding a row, (we just change the string here for simplicity). We’ll discuss creating fully customized cells in the next section.

```
public override UITableViewCell GetCell (UITableView tableView,
NSIndexPath indexPath)
{
    UITableViewCell cell;

    int row = indexPath.Row;

    if (row == _vc.Customers.Count) {
        cell = new UITableViewCell ();
        cell.TextLabel.Text = "Add Customer";

    } else {

        Customer aCustomer = _vc.Customers[row];

        if (String.IsNullOrEmpty (aCustomer.Note)) {
            cell = DequeueOrCreateCell (tableView,
                UITableViewCellStyle.Default, CUSTOMER_CELL);
        } else {
            cell = DequeueOrCreateCell (tableView,
                UITableViewCellStyle.Subtitle,
                CUSTOMER_CELL_WITH_NOTE);
            cell.DetailTextLabel.Text = aCustomer.Note;
        }

        cell.TextLabel.Text = String.Format ("{0} {1}",
            aCustomer.FName, aCustomer.LName);
    }
    return cell;
}
```

Since we support re-ordering of our customer rows, we need to account for that. Remove the ability to move the add row and prevent dropping beyond it by implementing CanMoveRow and CustomizeMoveTarget respectively in CustomersTableViewSource.

```
public override bool CanMoveRow (UITableView tableView, NSIndexPath
indexPath)
{
    return (indexPath.Row != _vc.Customers.Count);
}

public override NSIndexPath CustomizeMoveTarget (UITableView tableView,
NSIndexPath sourceIndexPath, NSIndexPath proposedIndexPath)
{
    NSIndexPath targetIndexPath;

    if (proposedIndexPath.Row == _vc.Customers.Count) {
        targetIndexPath =
            NSIndexPath.FromRowSection (proposedIndexPath.Row - 1, 0);
    } else {
        targetIndexPath = proposedIndexPath;
    }
    return targetIndexPath;
}
```

Finally, for the reason we did all this, adding a new customer. In your CommitEditingStyle implementation add a new customer object and insert a new row to present it by checking the editing style. When UITableViewEditingStyle is equal to Insert here, our new “Add Row” was selected and we can create the new customer. We’ll just set some arbitrary data on the customer for demonstration.

```
public override void CommitEditingStyle (UITableView tableView,
UITableViewCellEditingStyle editingStyle, NSIndexPath indexPath)
{
    // check if the edit operation was a delete
    if (editingStyle == UITableViewCellEditingStyle.Delete) {

        // remove the customer from the underlying data
        _vc.Customers.RemoveAt (indexPath.Row);

        // remove the associated row from the tableView
        tableView.DeleteRows (new NSIndexPath[] { indexPath },
            UITableViewRowAnimation.Middle);

    } else if (editingStyle == UITableViewCellEditingStyle.Insert) {

        _vc.Customers.Add (new Customer ("First", "Last"));
        tableView.InsertRows (new NSIndexPath[] {
            NSIndexPath.FromRowSection (_vc.Customers.Count - 1, 0) },
                UITableViewRowAnimation.None);
    }
}
```

There you have it. Run the app and select edit. You will have a new row in the table that you can select to create new customers interactively. (If only it were so easy to create new customers in the real world.)

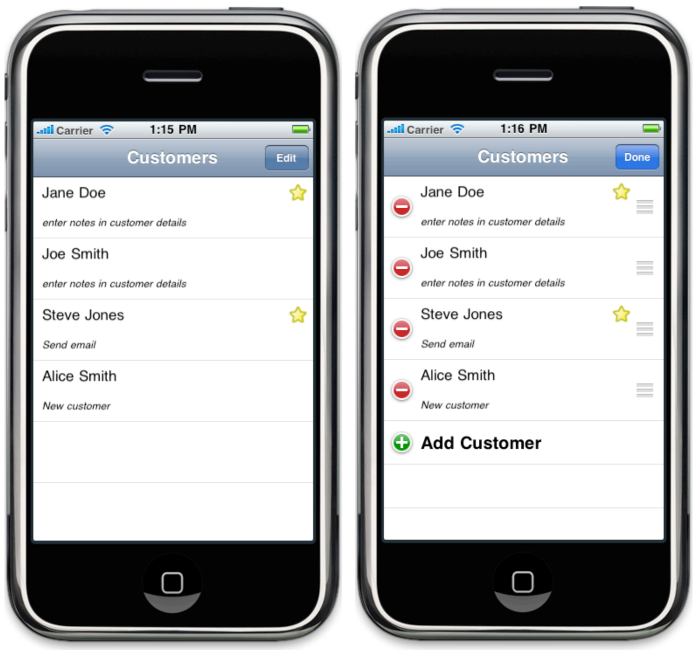
Customizing Tables Further With Custom Cells

The stock table styles are good for most common scenarios but sometimes you need complete control of the look of each cell. The good news is it’s pretty simple to derive from UITableViewCell to roll your own custom cell.

Let’s create a customer cell to display our customers. To make it a bit more interesting we’ll add a property to the Customer class to indicate if the customer is a favorite. Using this property, we’ll display a graphic in each cell for favorite customers (Figure 5.11).

Note

If the only customization were to display an image, setting the UITableViewCell.ImageView on either the default or subtitle cell styles would work fine. Even in this simple case however, a custom cell affords us complete control of things such as the layout.



***insert figure 5.11 05fig11.tif

Figure 5.11 UITableView with custom cells

The new Customer class is shown in Listing 5.6.

```
using System;

namespace LMT52
{
    public class Customer
    {
        string _fName;
        string _lName;

        public string FName {
            get { return this._fName; }
            set { _fName = value; }
        }

        public string LName {
            get { return this._lName; }
            set { _lName = value; }
        }

        public Customer (string fName, string lName)
        {
            _fName = fName;
            _lName = lName;
            IsFavorite = false;
        }

        public string Note { get; set; }

        public bool IsFavorite { get; set; }
    }
}
```

Listing 5.6 Customer class with IsFavorite property

To create a custom cell for presenting our customer data, we need to derive from UITableViewCell. We can create all the subviews that our cell will need in the subclasses' constructor. These views are added as subviews to the UITableViewCell's ContentView so that they will not interfere with the accessory view on the left or the cell re-ordering icon on the right. This way when the cell needs to be laid out for presentation, we can position everything relative to the ContentView.Bounds, which will adjust dynamically when editing to account for the UIAccessoryView.

When the cell layout is requested, the UITableView will call LayoutSubviews. Therefore, that is a good place to do exactly what the method name says, namely layout all the subviews. It is also where we can set any properties from our Customer object, which we can make a property of the cell class. Listing 5.7 shows our CustomerCell implementation.

```
public class CustomerCell : UITableViewCell
{
    UILabel _nameLabel;
    UILabel _noteLabel;
    UIImageView _newCustomerIcon;
    UIFont _noteFont;

    public Customer Customer { get; set; }

    public CustomerCell (Customer customer, string reuseIdentifier) :
        base(UITableViewCellStyle.Default, reuseIdentifier)
    {
        this.Customer = customer;

        _nameLabel = new UILabel ();
        _noteLabel = new UILabel ();
        _newCustomerIcon = new UIImageView ();
        _noteFont = UIFont.ItalicSystemFontOfSize(12.0f);

        this.ContentView.AddSubview (_nameLabel);
        this.ContentView.AddSubview (_noteLabel);
        this.ContentView.AddSubview (_newCustomerIcon);
    }

    public override void LayoutSubviews ()
    {
        base.LayoutSubviews ();
        _nameLabel.Text = String.Format ("{0} {1}", Customer.FName,
            Customer.LName);
        _noteLabel.Font = _noteFont;
        _noteLabel.Text = String.IsNullOrEmpty (Customer.Note) ?
            "enter notes in customer details" : Customer.Note;
        _newCustomerIcon.Image = Customer.IsFavorite ? UIImage.FromFile
            ("Favorite.png") : null;

        RectangleF b = ContentView.Bounds;

        float leftPadding = 10.0f;
        float rightPadding = 10.0f;
        float totalPadding = leftPadding + rightPadding;
        float iconWidth = b.Height / 2;
        float iconHeight = b.Height / 2;

        RectangleF nameRect = new RectangleF (b.Left + leftPadding,
            b.Top, b.Width/1.5f - totalPadding, b.Height / 2);

        _nameLabel.Frame = nameRect;

        RectangleF noteRect = new RectangleF (b.Left + leftPadding,
            b.Top + b.Height / 2, b.Width - totalPadding,
            b.Height / 2);

        _noteLabel.Frame = noteRect;

        RectangleF imageRect = new RectangleF (b.Right - iconWidth,
            b.Top, iconWidth, iconHeight);

        _newCustomerIcon.Frame = imageRect;
    }
}
```

Listing 5.7 CustomerCell class

Since we encapsulated setting user interface properties entirely in our cell, the calling code in CustomersTableViewSource can be simplified a bit in that it won't need to be responsible for setting the cell properties. The CustomerCell knows how to fill and present a customer. Listing 5.8 highlights the changes to CustomersTableViewSource.GetCell to use the new CustomerCell class.

```
public override UITableViewCell GetCell (UITableView tableView,
NSIndexPath indexPath)
{
    UITableViewCell cell;

    int row = indexPath.Row;

    if (row == _vc.Customers.Count) {
        cell = new UITableViewCell ();
        cell.TextLabel.Text = "Add Customer";
    } else {
        Customer aCustomer = _vc.Customers[row];
        cell = tableView.DequeueReusableCell (CUSTOMER_CELL);
        if (cell == null)
            cell = new CustomerCell(aCustomer, CUSTOMER_CELL);
        else
            (cell as CustomerCell).Customer = aCustomer;
    }

    return cell;
}
```

Listing 5.8 CustomersTableViewSource using CustomerCell class

You can also change the height of the rows in the CustomersTableViewSource and the CustomerCells will still layout appropriately since we made all the positioning code a function of the ContentView.Bounds (Listing 5.9).

```
public override float GetHeightForRow (UITableView tableView,
NSIndexPath indexPath)
{
    float h;
    if(indexPath.Row == _vc.Customers.Count){
        h = 50.0f;
    }
    else{
        h = 70.0f;
    }
    return h;
}
```

Listing 5.9 Implement GetHeightForRow to control cell(s) height

Adding Multiple Sections

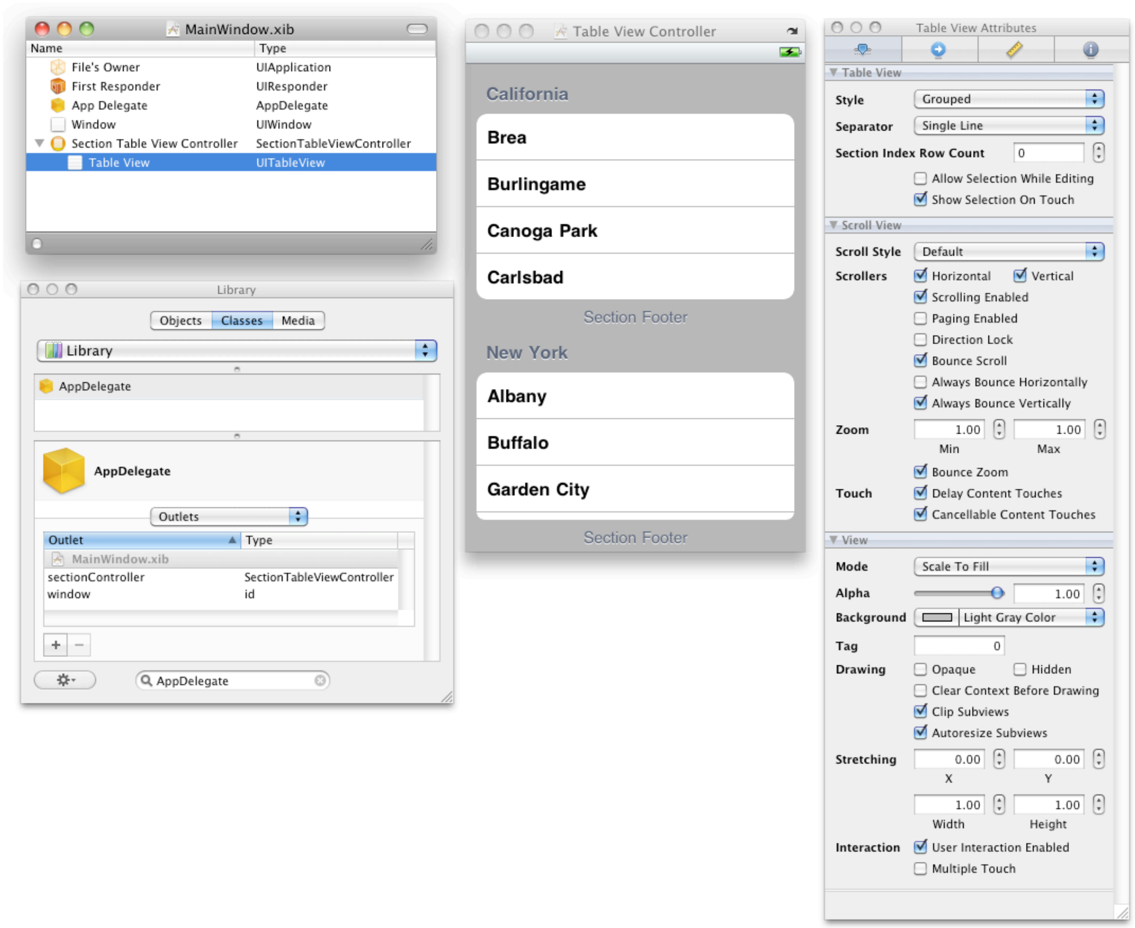
The examples we have used so contained a table with one section. UITableViews support creating multiple sections as well. Let's create a new example to demonstrate.

For this example let's use Interface Builder to create our UITableViewController so you can see how to set various properties of the table view there as well.

After creating a new iPhone project, open the MainWindow.xib in Interface Builder. First we will create our UITableViewController and configure it UITableView in IB, then we will go back to MonoDevelop to add the necessary code for populating the underlying models and creating the sections and rows. The steps to add the UITableViewController in IB are:

1. Add a TableViewController from the Library to MainWindow.xib
2. Set the class of the TableViewController to the name of the class will add in MonoDevelop, SectionTableViewController in this case.
3. Create and outlet named sectionController on the App Delegate of type SectionTableViewController
4. Connect the outlet to the SectionTableViewController we added in steps 1 and 2.
5. Set the UITableView style to Grouped.

Figure 5.12 shows Interface Builder after completing these steps as well as setting the TableView's background to something other than the default.



MainWindow.xib

Name	Type
File's Owner	UIApplication
First Responder	UIResponder
App Delegate	AppDelegate
Window	UIWindow
Section Table View Controller	SectionTableViewController
Table View	UITableView

Library

Objects | Classes | Media

Library

AppDelegate

AppDelegate

Outlets

Outlet	Type
MainWindow.xib	
sectionController	SectionTableViewController
window	id

AppDelegate

Table View Controller

California
Brea
Burlingame
Canoga Park
Carlsbad
Section Footer
New York
Albany
Buffalo
Garden City
Section Footer

Table View Attributes

Table View
Style: Grouped
Separator: Single Line
Section Index Row Count: 0
☐ Allow Selection While Editing
☑ Show Selection On Touch

Scroll View
Scroll Style: Default
Scrollers: ☑ Horizontal ☑ Vertical
☑ Scrolling Enabled
☐ Paging Enabled
☐ Direction Lock
☑ Bounce Scroll
☐ Always Bounce Horizontally
☑ Always Bounce Vertically
Zoom: 1.00 Min 1.00 Max
☑ Bounce Zoom
Touch: ☑ Delay Content Touches
☑ Cancellable Content Touches

View
Mode: Scale To Fill
Alpha: 1.00
Background: Light Gray Color
Tag: 0
Drawing: ☐ Opaque ☐ Hidden
☐ Clear Context Before Drawing
☑ Clip Subviews
☑ Autoresize Subviews
Stretching: 0.00 X 0.00 Y
1.00 Width 1.00 Height
Interaction: ☑ User Interaction Enabled
☐ Multiple Touch

***insert figure 5.12 05fig12.tif

Figure 5.12 Adding a UITableViewController in Interface Builder

Once you have everything added in Interface Builder, you need to switch over to MonoDevelop and add a class for the SectionTableViewController, making it a subclass of UITableViewController. Since this class is created form IB, you need to register it with the Objective-C runtime by decorating it with the RegisterAttribute. Also, you will need to supply a constructor that takes an IntPtr.

```
[Register("SectionTableViewController")]
public class SectionTableViewController : UITableViewController
{
    public SectionTableViewController (IntPtr p) : base (p) {}

    …
}
```

To implement a multiple sections, you need to override NumberSections in your UITableViewSource to return the desired number of sections and have GetCell return the cell for each section from the appropriate backing store, styled however you wish. In this example we use two string lists for simplicity, one for each section. Also, RowsInSections needs to return the appropriate row count for each section. Our implementation of SectionTableViewController is shown in Listing 5.10.

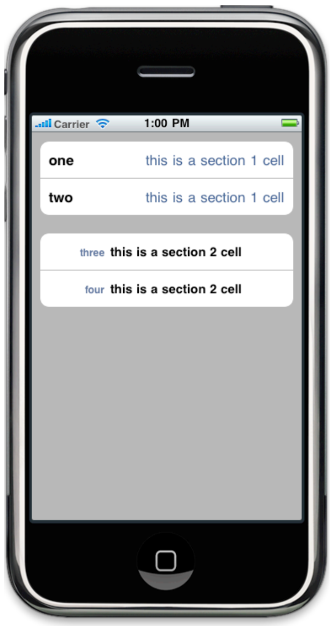
```
using System;
using System.Collections.Generic;
using MonoTouch.UIKit;
using MonoTouch.Foundation;

namespace LMT53
{
    [Register("SectionTableViewController")]
    public class SectionTableViewController : UITableViewController
    {
        public List<string> SectionOneList { get; set; }
        public List<string> SectionTwoList { get; set; }

        public SectionTableViewController (IntPtr p) : base (p) {}

        public override void ViewDidLoad ()
        {
            base.ViewDidLoad ();

            this.TableView.Source = new SectionSource (this);
        }

        class SectionSource : UITableViewSource
        {
            const string SECTION_ONE_CELL = "sectionOneCell";
            const string SECTION_TWO_CELL = "sectionTwoCell";

            SectionTableViewController _controller;

            public SectionSource (
                SectionTableViewController controller)
            {
                _controller = controller;
            }

            public override int NumberOfSections (
                UITableView tableView)
            {
                return 2;
            }

            public override int RowsInSection (
                UITableView tableview, int section)
            {
                if (section == 0) {
                    return _controller.SectionOneList.Count;
                } else {
                    return _controller.SectionTwoList.Count;
                }
            }

            public override UITableViewCell GetCell (
                UITableView tableView,
                MonoTouch.Foundation.NSIndexPath indexPath)
            {
                UITableViewCell cell;

                if (indexPath.Section == 0) {
                    cell = tableView.DequeueReusableCell (
                        SECTION_ONE_CELL);

                    if (cell == null)
                        cell = new UITableViewCell (
                            UITableViewCellStyle.Value1,
                            SECTION_ONE_CELL);

                    cell.TextLabel.Text =
                        _controller.SectionOneList[indexPath.Row];
                    cell.DetailTextLabel.Text =
                        "this is a section 1 cell";

                } else {
                    cell = tableView.DequeueReusableCell (
                        SECTION_TWO_CELL);

                    if (cell == null)
                        cell = new UITableViewCell (
                            UITableViewCellStyle.Value2,
                            SECTION_TWO_CELL);

                    cell.TextLabel.Text =
                        _controller.SectionTwoList[indexPath.Row];
                    cell.DetailTextLabel.Text =
                        "this is a section 2 cell";
                }

                return cell;
            }
        }
    }
}
```

Listing 5.10 UITableViewController supporting multiple sections

We can now make use of the SectionViewController. Recall we connected an outlet named sectionController from the AppDelegate to a SectionTableViewController, so we can use that in AppDelegate.FinishedLaunching to access the instance of SectionTableViewController, populate its lists with some sample data and add it to the window.

```
public override bool FinishedLaunching (UIApplication app, NSDictionary
options)
{
    List<string> list1 = new List<string> { "one", "two" };
    List<string> list2 = new List<string> { "three", "four" };
    sectionController.SectionOneList = list1;
    sectionController.SectionTwoList = list2;

    window.AddSubview (sectionController.View);
    window.MakeKeyAndVisible ();

    return true;
}
```

The resulting app is shown in Figure 5.13.


Carrier
1:00 PM
one this is a section 1 cell
two this is a section 1 cell
three this is a section 2 cell
four this is a section 2 cell

***insert figure 5.13 05fig13.tif

Figure 5.13 Grouped UITableView with multiple sections

MonoTouch.Dialog

As you have seen, the various classes that make up table driven user interfaces on the iPhone are quite extensive. In an effort to abstract the various mechanics involved in creating table-based applications, Miguel de Icaza created an incredible, open-source project called MonoTouch.Dialog. Although beyond the scope of this chapter, I wanted to at least give it a mention.

Creating a class and decorating it with various attributes allows you to easily create a table-based UI. For example the following class shows a few MonoTouch.Dialog defined attributes:

```
public class Customer
{
    [Section("Customer Name")]
    [Entry("Enter first name")]
    public string FirstName;

    [Entry("Enter last name")]
    public string LastName;

    [Section("More Customer Details")]

    [Entry("Enter customer note")]
    public string Note;

    [Checkbox]
    public bool IsFavorite = true;
}
```

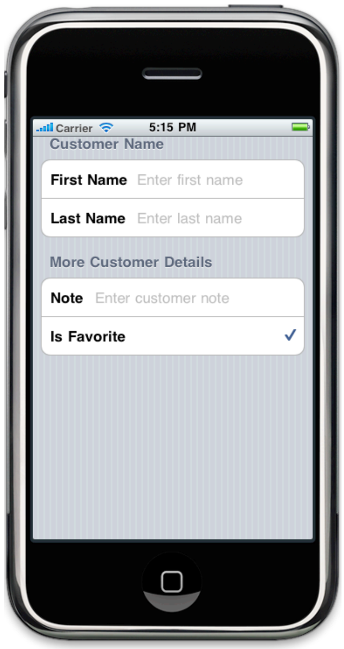
With MonoTouch.Dialog turning this class in a table-based UI takes three lines of code.

```
Customer c = new Customer ();
BindingContext b = new BindingContext (null, c, "Create a Customer");
DialogViewController dvc = new DialogViewController (b.Root);
```

Since the DialogViewController is a subclass of UITableViewController, its view is added to the user interface in the typical fashion.

```
window.AddSubview (dvc.View);
```

This results in the following user interface:



***insert figure 5.14 05fig14.tif

Figure 5.14 User Interface created with MonoTouch.Dialog

This example does not even scratch the surface of what you can accomplish with MonoTouch.Dialog. I encourage you to investigate this project further at http://github.com/migueldeicaza/MonoTouch.Dialog.

Where Are We?

We covered the fundamentals of creating table-driven user interfaces in this chapter. There are a plethora of capabilities supported with the UITableView, UITableViewController and UINavigationController classes that work together to create many of the application experiences you see on the iPhone. In the next chapter we will explore several system capabilities that enable additional experiences on the device.