Tables in iOS

Evolve Fundamentals Track, Chapter 3

Xamarin Inc.

Overview

This chapter will take a comprehensive look at tables and how to work with them. Topics covered will include:

- **Table parts** – Introducing and explaining the visual elements of the `UITableView` control.
- **Displaying data in tables** – Demonstrating how to create and populate a table, use different table and cell styles, and how to avoid memory issues by recycling cell objects.
- **Advanced Usage** – Building custom cells and using the editing features of the `UITableView` class.

The code samples are structured to incrementally add functionality as you progress through the chapter. There are seven sample projects included in the solution:

- **TablesiOS_demo1** – Implement a basic table
- **TablesiOS_demo2** – Add an index
- **TablesiOS_demo3** – Plain or Grouped style with a header
- **TablesiOS_demo4** – Alternate built-in cell layouts
- **TablesiOS_demo5** – Custom cell layouts
- **TablesiOS_demo6** – Using tables for navigation
- **TablesiOS_demo7** – Table editing features

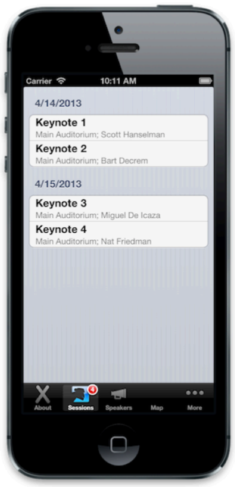
The code introduced in this chapter is already present in the samples, but commented out. Use the **Tasks** window in your IDE to quickly find the commented-out code that is marked with a `//TODO:` task identifier.

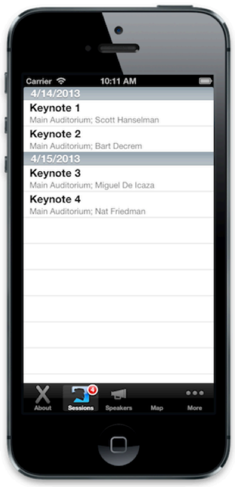
Table Parts & Functionality

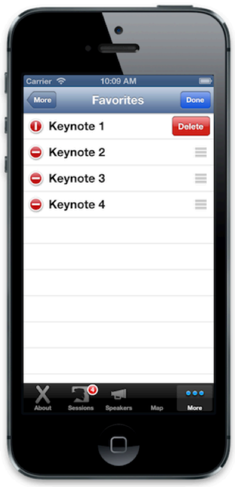
A `UITableView` can have a "grouped" or a "plain" style, and consists of the following parts:

- **Section Header** – Description or identifier for the group of rows that follow.
- **Cells** - Or rows, if you prefer.
- **Section Footer** – Summary text appearing below a group of rows.
- **Index** – A list of the different content groups in the table, and a mechanism that helps you scroll quickly to any row in the table.
- **Edit mode** – A change interface that includes "swipe to delete," and lets you change row order by dragging "handles."

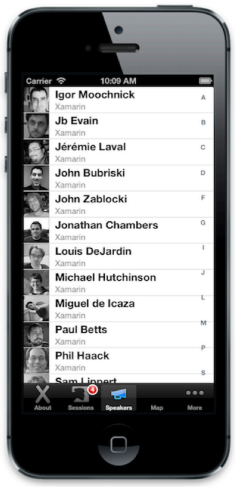
The following screenshots show how section rows, headers, footers, edit controls, and the index are displayed.


Carrier
10:11 AM
4/14/2013
Keynote 1
Main Auditorium; Scott Hanselman
Keynote 2
Main Auditorium; Bart Decrem
4/15/2013
Keynote 3
Main Auditorium; Miguel De Icaza
Keynote 4
Main Auditorium; Nat Friedman
About
Sessions
Speakers
Map
More
Grouped


Carrier
10:11 AM
4/14/2013
Keynote 1
Main Auditorium; Scott Hanselman
Keynote 2
Main Auditorium; Bart Decrem
4/15/2013
Keynote 3
Main Auditorium; Miguel De Icaza
Keynote 4
Main Auditorium; Nat Friedman
About
Sessions
Speakers
Map
More
Plain



Edit Mode



Index

These parts are described in more detail below:

Section Header

Cells can optionally be grouped into sections, and labeled with a custom header and/or footer. The header can be set with a string value or a custom view can be supplied to allow for a different layout or style.

Cells

Cells are the main user interface element for a table. When implemented correctly, cells are reused for memory efficiency. There are four built-in cell styles, and you can create your own custom cells as well.

Section Footer

The (optional) footer can be set with a string value or a custom view can be supplied to allow for a different layout or style. Section headers and footers can be set independently.

Index

The index appears as a strip of characters down the right edge of the table. Touching or dragging on the index accelerates scrolling to that part of the table. An index is optional, but it is recommended to help in navigating long lists. An index is not often used with the Grouped style.

Edit Features

There are a couple of different editing features available for tables:

- Swipe to delete individual cells.
- Entering Edit mode to reveal delete buttons on each row.

- Entering Edit mode to reveal reordering handles.
- Inserting new cells (with animation).

The remainder of this chapter shows how to use Xamarin.iOS to implement all these `UITableView` features.

Classes Overview

The primary classes used to display table views are as follows:

- UITableView – A view that contains a collection of cells inside a scrolling container. The table view typically uses the entire screen in an iPhone app, but may exist as part of a larger view on the iPad (or appear in a popover), or it may only use a small portion of the screen when appearing with other controls.
- UITableViewCell – A view that represents a single cell (or row) in a table view. There are four built-in cell types, and it is possible to create custom cells both in C# or with Interface Builder.
- UITableViewSource – A Xamarin.iOS-exclusive abstract class that provides all the methods required to display a table, including row count, returning a cell view for each row, handling row selection, and many other optional features. `UITableView` will not work unless you subclass `UITableViewSource`.
- NSIndexPath – This class contains row and section properties that uniquely identify the position of a cell in a table.
- UITableViewController – A ready-to-use `UIViewController` that has a `UITableView` hardcoded as its view. It's accessible via the TableView property.
- UIViewController – If the table does not occupy the entire screen, then you can add a `UITableView` to any `UIViewController` with its Frame set.

`UITableViewSource` encapsulates the following two classes, which are still available in Xamarin.iOS. They are not normally required:

- UITableViewDataSource – An Objective-C protocol that is modeled in MonoTouch as an abstract class. Must be subclassed to provide a table with a view for each cell, and in order to provide information about headers, footers, and the number of rows and sections in the table. Protocols are similar to interfaces in C# except that members can be optional. You can read more about protocols in the Protocols section below.
- UITableViewDelegate – An Objective-C protocol that is modeled in MonoTouch as a class. Handles selection, editing features, and other optional table features.

In this course, the examples all use `UITableViewSource` and ignore these two classes. They are mentioned here because any Objective-C examples found in

Apple's documentation will reference them, so it is useful to understand what they do (and that you can use MonoTouch's `UITableViewSource` instead).

Protocols

A protocol is an Objective-C language feature that provides a list of method declarations. It serves a similar purpose to an interface in C#, the main difference being that a protocol can have optional methods. Optional methods are not called if the class that adopts a protocol does not implement them. Also, a single class in Objective-C can implement multiple protocols, just as a C# class can implement multiple interfaces.

How Protocols are used with Delegates

Protocols provide a known set of methods for classes to call after certain events occur, such as after the user selecting a cell in a table. The classes that implement these methods are known as the delegates (not to be confused with a C# delegate) of the classes that call them.

Classes that support delegation do so by exposing a `Delegate` property, to which a class implementing the delegate is assigned. The methods you implement for the delegate will depend upon the protocol that the particular delegate adopts. For the `UITableView` method, you implement the `UITableViewDelegate` protocol.

The methods in the `UITableViewDelegate` allow handling interaction that occurs between the user and then `UITableView`. For convenience Xamarin.iOS combines the `UITableViewDelegate` along with the `UITableViewDataSource` in a single class called `UITableViewSource`. `UITableViewDataSource` is the class that Xamarin.iOS binds to the `UITableViewDataSource` protocol. The methods found in `UITableViewDataSource`, which again are made available in `UITableViewSource`, allow the table to be populated with data.

Populating a Table with Data

To add rows to a `UITableView`, you need to implement a `UITableViewSource` subclass, and then override the methods that the table view calls to populate itself.

Subclassing UITableViewSource

A `UITableViewSource` subclass is assigned to every `UITableView`. The table view queries the source class to determine how to render itself (for example, it may ask how many rows are required and the height of each row, if different from the default). Most importantly, the source supplies each cell view populated with data.

There are only two mandatory methods that are required in order to make a table display data:

- RowsInSection – Return an `int` count of the total number of rows of data that the table should display.

- **GetCell** – Return a `UITableCellView` populated with data for the corresponding row index that is passed to the method.

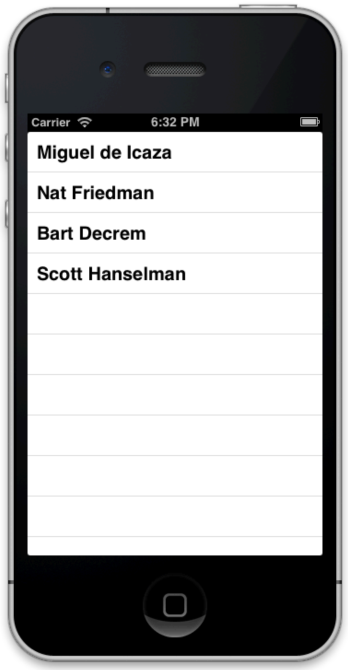
The `SpeakersTableSource` class shown below (from the **TablesiOS_demo1** project) has the simplest possible implementation of `UITableViewSource`. It accepts an array of strings to display in the table and returns a default cell style that contains each string:

```
class SpeakersTableSource : UITableViewSource{
    static readonly string speakerCellId = "SpeakerCell";
    protected string[] speakers;

    public SpeakersTableSource (string[] speakers)
    {
        speakers = speakers;
    }
    public override int RowsInSection (UITableView tableview, int section)
    {
        return speakers.Length;
    }
    public override UITableViewCell GetCell (UITableView tableView,
NSIndexPath indexPath)
    {
        UITableViewCell cell = tableView.DequeueReusableCell
(speakerCellId);

        if (cell == null)
            cell = new UITableViewCell (UITableViewCellStyle.Default,
speakerCellId);

        cell.TextLabel.Text = speakers [indexPath.Row];
        return cell;
    }
}
```

To use this subclass, the `SpeakersViewController` class creates a string array to construct the source, and then assigns it to a `UITableView` instance. The `SpeakersViewController` `ViewDidLoad` method looks like this:

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    string[] speakers = new string[] {
        "Miguel De Icaza",
        "Nat Friedman",
        "Bart Decrem",
        "Scott Hanselman"
    };
    TableView.Source = new SpeakersTableSource (speakers);
}
```

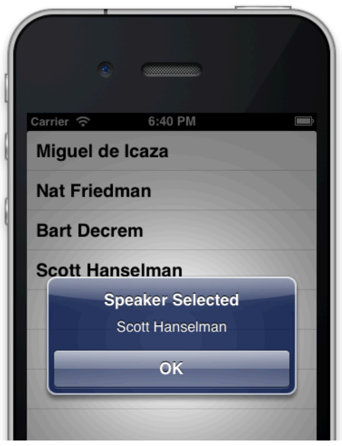
The resulting table looks like this:



Most tables allow the user to touch a row to select it, and then perform some other action (such as playing a song, calling a contact, or showing another screen). To respond to user touches, the `RowSelected` method needs to be implemented, as shown below:

```
public override void RowSelected (UITableView tableView, NSIndexPath
indexPath)
{
    new UIAlertView ("Speaker Selected", speakers [indexPath.Row], null,
"OK", null).Show ();

    tableView.DeselectRow (indexPath, true);
}
```

Now the user can touch a row and an alert will appear:



Reusing Table Cells

There are only four items in this example. The screen can fit ten, so no cell reuse is required. When displaying hundreds or thousands of rows and when only ten fit on the screen at a time, it would be a waste of memory to create hundreds or thousands of `UITableViewCell` objects. To avoid this situation, when a cell disappears from the screen, its view is placed in a queue for reuse. As the user scrolls, the table calls `GetCell` to request new views to display. To reuse an existing cell that is not currently being displayed, simply call the `DequeueReusableCell` method. If a cell is available for reuse, then it will be returned, otherwise a null is returned and your code must create a new cell instance.

This snippet of code from the example demonstrates the pattern:

```
// request a recycled cell to save memory
UITableViewCell cell = tableView.DequeueReusableCell (speakerCellId);

// if there are no cells to reuse, create a new one
if (cell == null)
    cell = new UITableViewCell (UITableViewCellStyle.Default,
cellIdentifier);
```

The `speakerCellId` effectively creates separate queues for different types of cells. In this example, all the cells look the same so only one hardcoded identifier is used. The cell identifier is the same string for each type of cell – in our example it is a constant value.

If there were different types of cells, then they should each have a different identifier string (one for each type of cell, NOT one for each individual cell), both when they are instantiated and when they are requested from the reuse queue.

Reloading Data

Sometimes the data in a table needs to be updated. This could happen if the user hits a refresh button or enters new data on another screen, or if a background thread (such as a web request) retrieves new information to display.

There are two steps required to display new data:

1. Update the `UITableViewSource` with the new data.
2. Call `ReloadData()` on the `UITableView` instance to display it.

The following sample code demonstrates table data being reloaded (based on the previous sample). The first change in the sample is a new method on the `TableSource` class that changes the underlying data to be displayed:

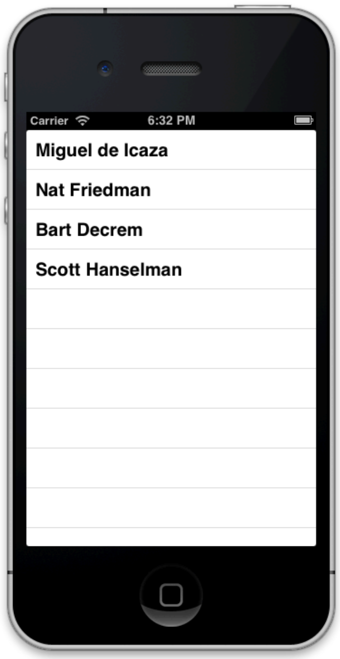
```
public void UpdateSpeakers (string[] items)
{
    this.speakers = speakers;
}
```

The second change (in the `SpeakerViewController` class) will cause the new data to be displayed by updating the `TableSource` with a new array of strings, and then calling the `ReloadData` method.

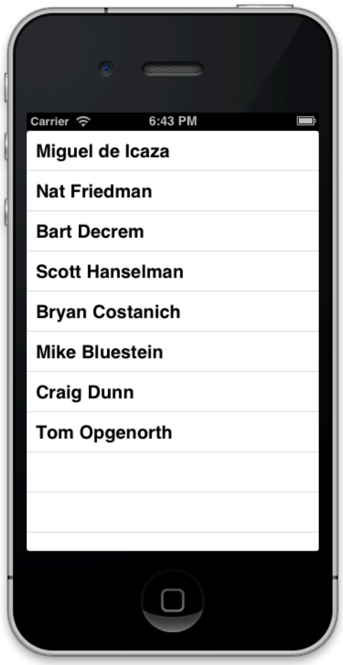
For this example, a simple timer has been used to demonstrate how to update the table's data after a ten second delay:

```
NSTimer.CreateScheduledTimer (TimeSpan.FromSeconds (10), delegate {
    string[] updatedSpeakers = new string[] {
        "Scott Hanselman",
        "Bart Decrem",
        "Miguel De Icaza",
        "Nat Friedman",
        "Bryan Costanich",
        "Mike Bluestein",
        "Craig Dunn",
        "Tom Opgenorth"
    };
    ((SpeakersTableSource)TableView.Source).UpdateSpeakers
(updatedSpeakers);

    TableView.ReloadData ();
});
```

Typically, `ReloadData` will be called in response to a user interaction (such as a button press or a network request completion). These screenshots show the table before and after the data is reloaded:



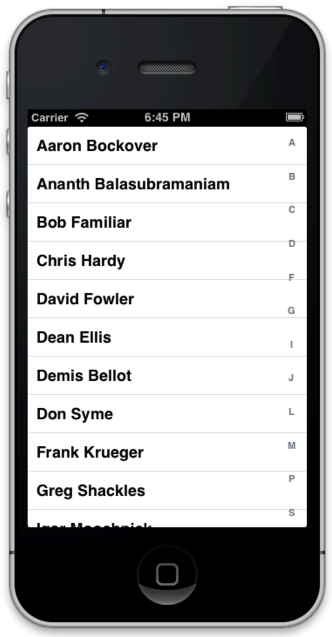
Miguel de Icaza
Nat Friedman
Bart Decrem
Scott Hanselman



Miguel de Icaza
Nat Friedman
Bart Decrem
Scott Hanselman
Bryan Costanich
Mike Bluestein
Craig Dunn
Tom Opgenorth

Adding an Index

An index helps the user scroll through long lists, typically ordered alphabetically, although you can index by whatever criteria you wish. In order to demonstrate the index, the sample that accompanies this chapter loads a much longer list of items from a file. Each item in the index corresponds to a section of the table, as shown below:



Speaker Class

For the TablesiOS_demo2 project we have created a `Speaker` class instead of a string to represent each speaker. The `Speaker` class is shown below:

```
public class Speaker
{
    public string HeadshotUrl { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public Speaker () {}
}
```

To support "sections," the data behind the table needs to be grouped, so the sample creates an `IGrouping<K,T>[]` from the List by using the first letter of each item as the key and `Speaker` as the type. Then as the user selects indices in the table, the speakers for the index section can be easily obtained.

In addition to using a data structure that supports grouping, the following methods must be added or modified to the `UITableViewSource` subclass:

- NumberOfSections – This method is optional. By default, the table assumes one section. When displaying an index, this method should return the number of items in the index (for example, "26" will be returned if the index contains all the letters of the English alphabet). In our sample, this method returns the number of speaker indices.
- RowsInSection – This method returns the number of rows in a given section.
- SectionIndexTitles – This method returns the array of strings that will be used to display the index. The sample code returns an array of letters.

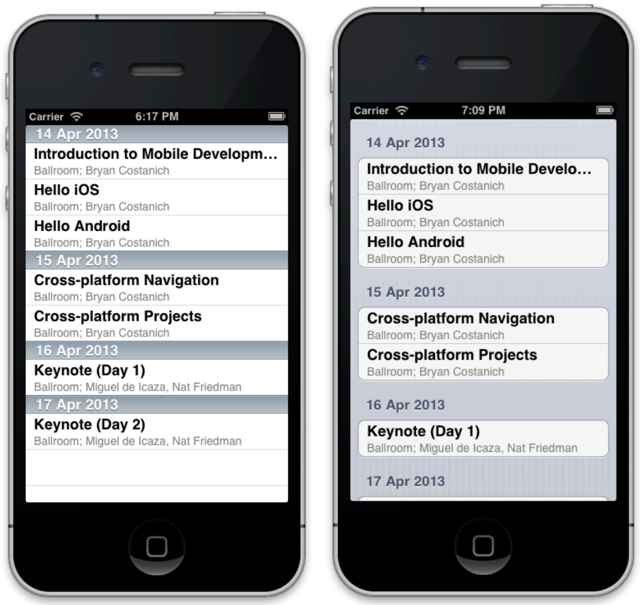
The updated methods in the `SpeakersTableSource` class look like this:

```
public class SpeakersTableSource : UITableViewSource
{
    // cell identifier (used for cell reuse)
    static readonly string speakerCellId = "SpeakerCell";
    // list of speakers
    List<Speaker> data;
    // an index of all first letters
    string[] indices;
    // speakers grouped by their first letter
    IGrouping<char,Speaker>[] grouping;

    public SpeakersTableSource (List<Speaker> speakers)
    {
        data = speakers;
        indices = SpeakerIndicies ();
        grouping = GetSpeakersGrouped();
    }
    public override int NumberOfSections (UITableView tableView)
    {
        return indices.Count ();
    }
    public override int RowsInSection (UITableView tableview
              , int section)
    {
        return grouping [section].Count ();
    }
    public override string[] SectionIndexTitles (UITableView tableView)
    {
        return indices;
    }
    public string[] SpeakerIndicies ()
    {
        var indicies = (from s in data
                    orderby s.Name ascending
                    group s by s.Name [0] into g
                    select g.Key.ToString ()).ToArray ();

        return indicies;
    }
    IGrouping<char, Speaker>[] GetSpeakersGrouped ()
    {
        var speakersGrouped = (from s in data
                           orderby s.Name ascending
                           group s by s.Name [0] into g
                           select g).ToArray ();

        return speakersGrouped;
    }
}
```

Indexes are generally only used with the `Plain` table style.

Adding Headers and Footers

Headers and footers can be used to visually group rows in a table. The data structure required is very similar to the structure used when adding an index, as are the methods to implement in the `UITableViewSource` subclass. Instead of using the alphabet to group speaker cells however, this example will instead group sessions by day. The output looks like this (both `Plain` and `Grouped` table styles are shown).



14 Apr 2013
Introduction to Mobile Developm…
Ballroom; Bryan Costanich
Hello iOS
Ballroom; Bryan Costanich
Hello Android
Ballroom; Bryan Costanich
15 Apr 2013
Cross-platform Navigation
Ballroom; Bryan Costanich
Cross-platform Projects
Ballroom; Bryan Costanich
16 Apr 2013
Keynote (Day 1)
Ballroom; Miguel de Icaza, Nat Friedman
17 Apr 2013
Keynote (Day 2)
Ballroom; Miguel de Icaza, Nat Friedman

14 Apr 2013
Introduction to Mobile Develo…
Ballroom; Bryan Costanich
Hello iOS
Ballroom; Bryan Costanich
Hello Android
Ballroom; Bryan Costanich
15 Apr 2013
Cross-platform Navigation
Ballroom; Bryan Costanich
Cross-platform Projects
Ballroom; Bryan Costanich
16 Apr 2013
Keynote (Day 1)
Ballroom; Miguel de Icaza, Nat Friedman
17 Apr 2013

In the TablesiOS_demo3 example, the following class represents a session:

```
public class Session
{
    public string Speaker { get; set; }
    public string Title { get; set; }
    public string Abstract { get; set; }
    public string Location { get; set; }
    public DateTime Begins { get; set; }
    public DateTime Ends { get; set; }
    public Session () {}
}
```

To display headers and footers, the `UITableViewSource` subclass requires these additional methods:

- TitleForHeader – Returns the text that will be used as the header.
- TitleForFooter – Returns the text that will be used as the footer.

The example uses only the header, to display the day for the session group. It does this by implementing the `TitleForHeader` method `SessionsTableSource` class as shown below:

```
public override string TitleForHeader (UITableView tableView, int section)
{
    return grouping [section].ElementAt(0)
```

```
                    .Begins.Date.ToShortDateString ();
}
```

You can further customize the appearance of the header and footer with a View object. Using the `GetViewForHeader` and `GetViewForFooter` method overrides on `UITableViewSource`.

Customizing the Table's Appearance

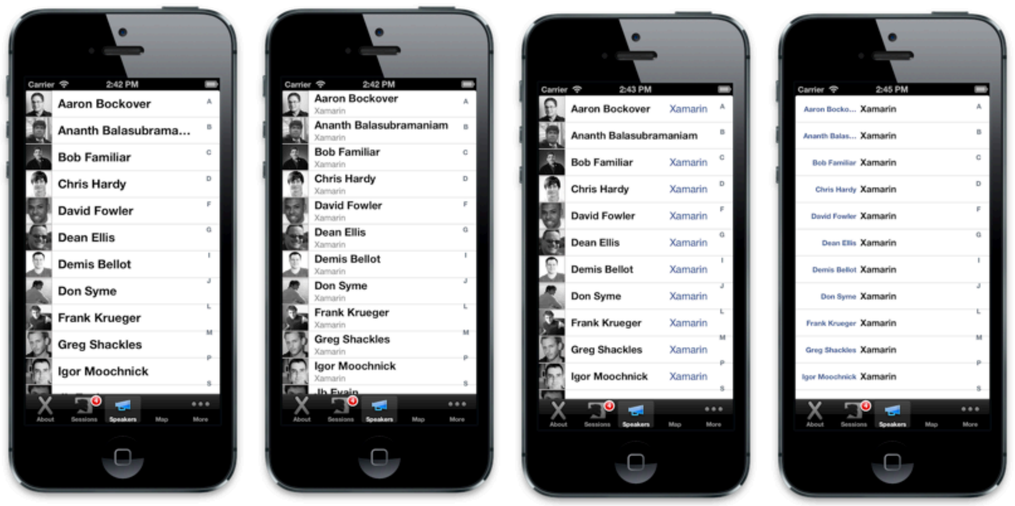
The simplest way to change the appearance of a table is to use a different cell style. When you create a cell in the `UITableViewSource`'s `GetCell` method, you can determine which cell style is used. The **TablesiOS_demo4** sample project demonstrates how to change cell styles.

UITableViewCell Styles

There are four built-in styles:

- **Default** – Supports a `UIImageView`.
- **Subtitle** – Supports a `UIImageView` and a subtitle.
- **Value1** – Right-aligned subtitle. Supports a `UIImageView`.
- **Value2** – Title is right aligned and subtitle is left aligned (but has no image).

The following screenshots show how each style appears:



Default   Subtitle   Value1   Value2

To produce these screens, the cell style is set in the `UITableViewCell` constructor in the `SpeakersTableSource` class, like this:

```
cell = new UITableViewCell (UITableViewCellStyle.Default, speakerCellId);
//cell = new UITableViewCell (UITableViewCellStyle.Subtitle,
speakerCellId);
//cell = new UITableViewCell (UITableViewCellStyle.Value1, speakerCellId);
//cell = new UITableViewCell (UITableViewCellStyle.Value2, speakerCellId);
```

The cell's properties can then be set (do not set properties that the style does not support or an exception will be thrown):

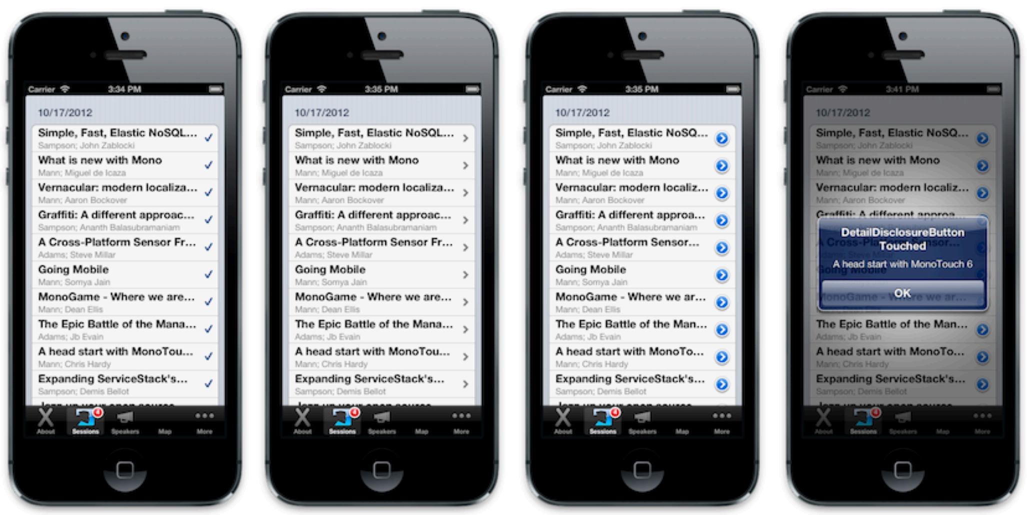
```
cell.TextLabel.Text = speaker.Name;
cell.DetailTextLabel.Text = speaker.Company;
cell.ImageView.Image = speaker.Image; //don't use for Value2
```

Accessories

Cells can also have the following accessories added to the right of the view:

- Checkmark – Can be used to indicate selection in a table.
- DisclosureIndicator – Normally used to indicate that touching the cell will open another view.
- DetailDisclosureIndicator – Responds to touch independently of the rest of the cell, allowing it to perform a different function than is triggered by a direct touch of the cell itself.

The following image shows how each of these accessories appears in the table:


10/17/2012
Simple, Fast, Elastic NoSQL...
What is new with Mono
Vernacular: modern localiza...
Graffiti: A different approac...
A Cross-Platform Sensor Fr...
Going Mobile
MonoGame - Where we are...
The Epic Battle of the Mana...
A head start with MonoTou...
Expanding ServiceStack's...
DetailDisclosureButton Touched
A head start with MonoTouch 6
OK

Checkmark DisclosureIndicator DetailDisclosureButton (Tapped)

To display one of these accessories, you can set the `Accessory` property in the `GetCell` method:

```
cell.Accessory = UITableViewCellAccessory.Checkmark;
//cell.Accessory = UITableViewCellAccessory.DisclosureIndicator;
//cell.Accessory = UITableViewCellAccessory.DetailDisclosureButton; //
implement AccessoryButtonTapped
//cell.Accessory = UITableViewCellAccessory.None; // to clear the
accessory
```

When the `DetailDisclosureButton` is shown, you should also override the `AccessoryButtonTapped` to perform some action when it is touched, as shown in the following code:

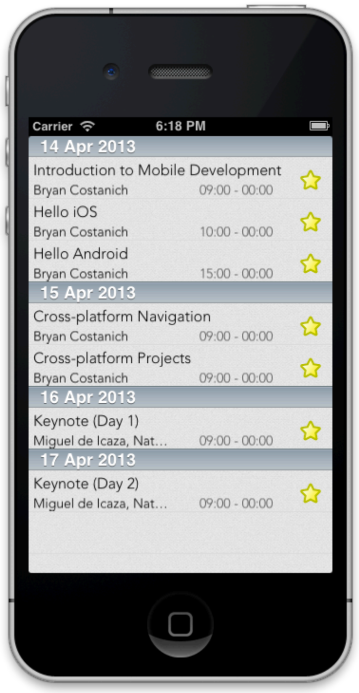
```
public override void AccessoryButtonTapped (UITableView tableView,
NSIndexPath indexPath)
```

```
{
    var speaker = data.ElementAt (indexPath.Row);

    new UIAlertView ("Session Accessory Tapped", speaker.Name, null, "OK"
                , null).Show ();
}
```

Creating Custom Cell Layouts

To change the visual style of a table to something other than what is offered by the system-provided cell styles, you need to supply custom cells for it to display. The custom cells can have additional customizations such as different colors and control layout.

The `SessionCell` class implements a `UITableViewCell` subclass that defines a custom layout of `UILabels`, with different fonts and colors, a `UIImage` and a gradient background drawn with Core Graphics. The resulting cells look like this:


Carrier
6:18 PM
14 Apr 2013
Introduction to Mobile Development
Bryan Costanich 09:00 - 00:00
Hello iOS
Bryan Costanich 10:00 - 00:00
Hello Android
Bryan Costanich 15:00 - 00:00
15 Apr 2013
Cross-platform Navigation
Bryan Costanich 09:00 - 00:00
Cross-platform Projects
Bryan Costanich 09:00 - 00:00
16 Apr 2013
Keynote (Day 1)
Miguel de Icaza, Nat… 09:00 - 00:00
17 Apr 2013
Keynote (Day 2)
Miguel de Icaza, Nat… 09:00 - 00:00

The custom cell class consists of only two methods and one property:

- Constructor – Creates the UI controls and sets the custom style properties (for example: font face, size, and colors).
- LayoutSubviews – Sets the location of the UI controls. In the example, every cell has the same layout, but a more complex cell (particularly those with varying sizes) might need different layout positions, depending on the content being displayed. Drawing the gradient background is implemented in LayoutSubviews as well.
- Session – Property used to pass data to be displayed by the cell.

The complete sample code for the `SessionCell` class in the TablesiOS_demo5 project is as follows:

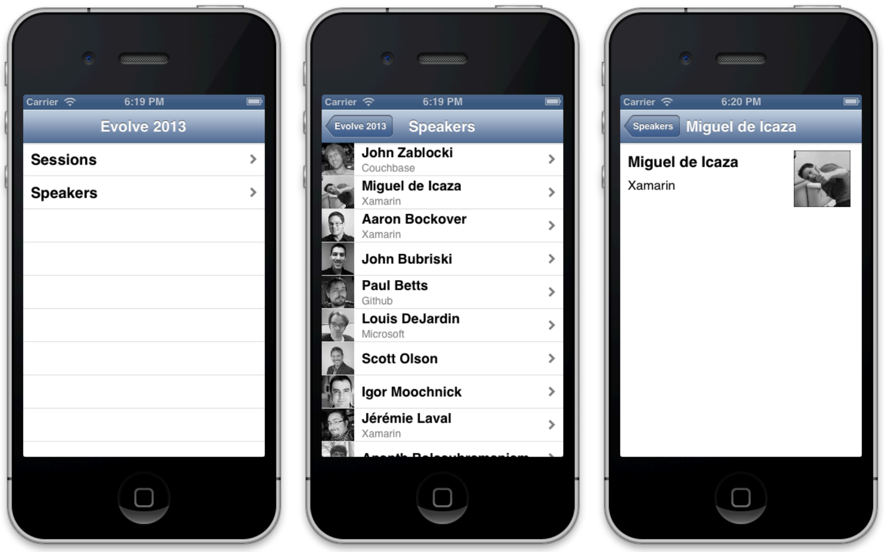
```
public class SessionCell : UITableViewCell {
    UILabel titleLabel, timeLabel, speakerLabel;
    UIImageView favoriteImageView;

    public Session Session { get; set; }

    public SessionCell (string reuseIdentifier):
base(UITableViewCellStyle.Default, reuseIdentifier)
    {
        titleLabel = new UILabel { Font = UIFont.FromName("Avenir-Light",
16.0f ), BackgroundColor = UIColor.Clear  };
        timeLabel = new UILabel { Font = UIFont.FromName("Avenir-Light",
14.0f ), TextColor = UIColor.DarkGray, BackgroundColor = UIColor.Clear };
        speakerLabel = new UILabel { Font = UIFont.FromName("Avenir-Light",
14.0f ), BackgroundColor = UIColor.Clear  };

        ContentView.AddSubview (titleLabel);
        ContentView.AddSubview (timeLabel);
        ContentView.AddSubview (speakerLabel);

        favoriteImageView = new UIImageView ();
        ContentView.AddSubview (favoriteImageView);
    }

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

        float padding = 5.0f;

        titleLabel.Text = Session.Title;
        timeLabel.Text = String.Format ("{0} - {1}",
Session.Begins.ToString ("HH:mm"), Session.Ends.ToString ("HH:mm"));
        speakerLabel.Text = (Session.Speaker != null) ? Session.Speaker :
"";

        RectangleF b = ContentView.Bounds; // ContentView.Bounds can
change! see Accessories or Editing mode

        var titleRect = new RectangleF (b.Left + padding, b.Top + padding,
b.Width - 2 * padding, (b.Height / 3) + 4);
        titleLabel.Frame = titleRect;

        var speakerRect = new RectangleF (b.Left + padding,
titleRect.Bottom + padding, b.Width/2 - 2 * padding, b.Height/3);
        speakerLabel.Frame = speakerRect;

        var timeRect = new RectangleF (speakerRect.Right + 5 * padding,
titleRect.Bottom + padding, b.Width/2 - 2 * padding, b.Height/3);
        timeLabel.Frame = timeRect;

        favoriteImageView.Image = UIImage.FromBundle ("images/favorited");
        favoriteImageView.Frame = new RectangleF (b.Width - 42, 5, 38,
38);
    }
}
```

To create custom cells using the `SessionCell` class, the `GetCell` method of the `UITableViewSource` needs to be modified as follows:

```
public override UITableViewCell GetCell (UITableView tableView,
NSIndexPath indexPath)
{
    var sessionGroup = this.sessionsGrouped [indexPath.Section];
    var session = sessionGroup.ElementAt (indexPath.Row);

    var cell = (SessionCell) tableView.DequeueReusableCell
(sessionCellId);
    if (cell == null)
        cell = new SessionCell (speakerCellId);

    cell.Session = session;

    return cell;
}
```

Using Tables for Navigation

A common use for the `UITableViewController` is to help users navigate through hierarchical lists of data, especially when used in conjunction with a `UINavigationController`. The **TablesiOS_demo6** sample shows how to combine these two controls to build a simple hierarchical menu system as shown in these screenshots:



The first two screens are tables and the final screen is a custom `UIViewController`. This isn't an in-depth look at the `UINavigationController` class, just a quick example of how it can be used in conjunction with `UITableViews`.

Menu screen

Similar to the first example in this chapter, the menu screen is driven by a hardcoded array of strings.

```
string[] items = new string[] {"Sessions", "Speakers"};
TableView.Source = new MenuTableSource (items, this);
```

To make the hierarchical navigation work, we load this screen into a `UINavigationController` in the projects' `AppDelegate` class. The `FinishedLaunching` method looks like this:

```
public override bool FinishedLaunching (UIApplication app, NSDictionary
options)
{
    evolveNavigationController = new UINavigationController ();
    evolveNavigationController.PushViewController (new
MenuTableViewController (), false);
    window = new UIWindow (UIScreen.MainScreen.Bounds);
    window.MakeKeyAndVisible ();
    window.RootViewController = evolveNavigationController;
    return true;
}
```

Once the menu is pushed into a `UINavigationController`, subsequent screens can also be pushed on, like a stack. iOS automatically adds the back button to make it easy for users to move back and forth through the options.

Speakers Screen

The Speakers list is the same `UITableViewController` that we used in TablesiOS_demo4, except that the `RowSelected` method creates the speaker details screen and pushes in onto the navigation stack.

```
public override void RowSelected (UITableView tableView, NSIndexPath
indexPath)
{
    var speaker = data [indexPath.Row];
    controller.NavigationController.PushViewController (new
SpeakerViewController (speaker), true);
    tableView.DeselectRow (indexPath, true);
}
```

Speaker Details Screen

The speaker details screen is a simple `UIViewController`, using C# code to layout the user interface controls. The speaker object is passed via the constructor and its properties are rendered in the `ViewDidLoad` method.

```
public class SpeakerViewController : UIViewController {
    Speaker speaker;
    public SpeakerViewController (Speaker showSpeaker)
    {
        speaker = showSpeaker;
    }
    UILabel name, company;
```

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

            Title = speaker.Name;
            View.BackgroundColor = UIColor.White;

            name = new UILabel (new RectangleF(10, 10, 200, 30));
            name.Font = UIFont.BoldSystemFontOfSize (20f);
            company = new UILabel (new RectangleF( 10, 40, 200, 30));
            avatar = new UIImageView (new RectangleF (230, 10, 75, 75));

            View.Add (name);
            View.Add (company);
            View.Add (avatar);

            name.Text = speaker.Name;
            company.Text = speaker.Company;
            avatar.Image = UIImage.FromBundle (speaker.HeadshotUrl);
        }
    }
```

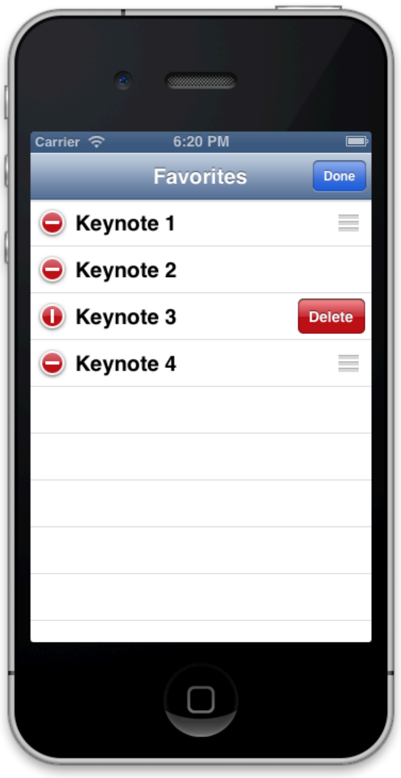
Editing

Overriding methods in a `UITableViewSource` subclass enables table-editing features. The simplest editing behavior is the swipe-to-delete gesture that can be implemented with a single method override. More complex editing (including moving rows) can be done when the table is in edit mode.

These features are demonstrated in the TablesiOS_demo7 sample project.

Swipe to Delete

The following screenshot from the `FavoritesViewController` class in the sample shows a cell with swipe-to-delete functionality:



There are three method overrides that affect the swipe gesture and show a **Delete** button in a cell:

- **CommitEditingStyle** – The table source detects whether or not this method is overridden and automatically enables the swipe-to-delete gesture. The method's implementation should call `DeleteRows` on the `UITableView` to cause the cells to disappear, and also remove the underlying data from the model (for example: an array, dictionary, or database).
- **CanEditRow** – (Optional) If `CommitEditingStyle` is not overridden, all rows are assumed to be editable. If this method is implemented and returns `false` (for some specific rows, or for all rows), then the swipe-to-delete gesture will not be available in that cell.
- **TitleForDeleteConfirmation** – (Optional) Specifies the text for the **Delete** button. If this method is not implemented, the button text will be "Delete."

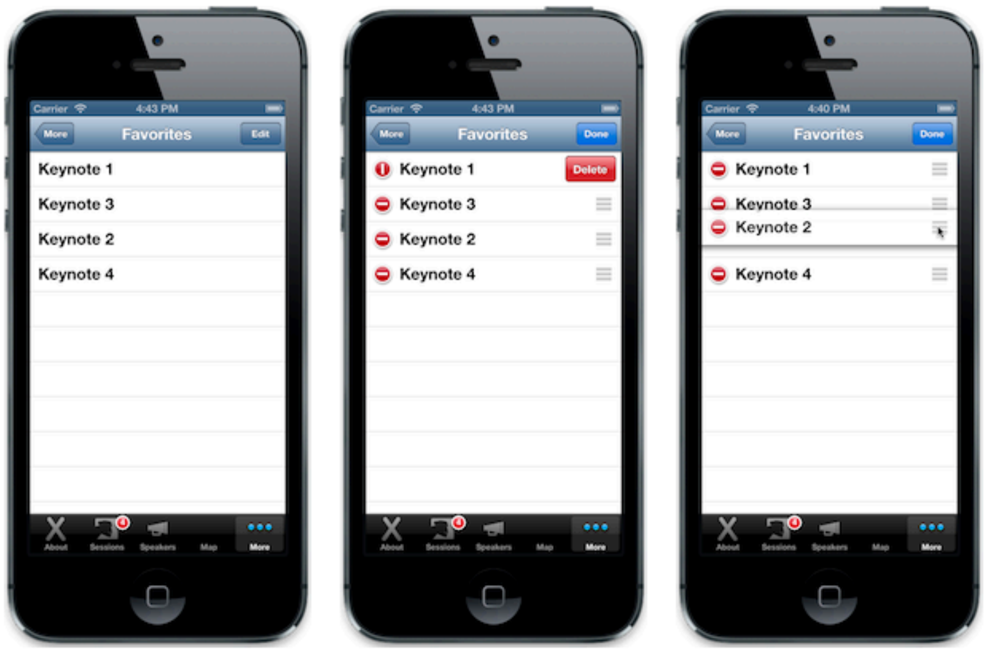
The following code from the `FavoritesViewController` class shows the `CommitEditingStyle` implementation that handles removing the cell and the underlying session data:

```
public override void CommitEditingStyle (UITableView tableView,
UITableViewCellEditingStyle editingStyle, NSIndexPath indexPath)
{
    if (editingStyle == UITableViewCellEditingStyle.Delete) {


        sessions.RemoveAt (indexPath.Row);
```

```
            tableView.DeleteRows (new NSIndexPath[] { indexPath },
UITableViewRowAnimation.Middle);
        }
    }
```

Edit Mode

When a table is in edit mode, the user sees a red "stop" widget on each row, which reveals a Delete button when touched. The table also displays a "handle" icon to indicate that the row can be dragged to change the order. The `FavoritesViewController` class implements these features as shown:



There are a number of different methods on `UITableViewSource` that affect a table's edit mode behavior:

- CanEditRow – (Optional) Whether each row can be edited. Return `false` to prevent both swipe-to-delete and deletion while in edit mode. Defaults to `true` if not overridden.
- CanMoveRow – (Optional) Return `true` to enable the move "handle" or `false` to prevent moving. Defaults to `true` if not overridden.
- EditingStyleForRow – (Optional) When the table is in edit mode, the return value from this method determines whether the cell displays a red deletion icon or a green add icon. Return `UITableViewCellEditingStyle.None` if the row should not be editable. Defaults to UITableViewCellEditingStyle.Delete if not overridden.
- MoveRow – Called when a row is moved so that the underlying data structure can be modified to match the data as it is displayed in the table.

The FavoritesViewController class uses the default implementations of CanEditRow, CanMoveRow and EditingStyleForRow respectively, making all rows support both moving and deleting.

The MoveRow implementation needs to alter the underlying data structure to match the new order, as shown below:

```
public override void MoveRow (UITableView tableView, NSIndexPath
sourceIndexPath, NSIndexPath destinationIndexPath)
{
    Session s = sessions [sourceIndexPath.Row];
    sessions.RemoveAt (sourceIndexPath.Row);
    sessions.Insert (destinationIndexPath.Row, s);
}
```

Finally, to get the table into edit mode, the navigation bar needs to include the EditButtonItem as shown:

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

When the user is finished editing, the Done button turns off the editing mode automatically.

Summary

The UITableView class provides a flexible way to present data, whether it is a short menu, a long scrolling list, or a detail view/input form. It provides usability features like the index and editing mode, and programming features like cell re-use that help keep memory usage low.

This chapter introduced some simple examples that used the default appearance, and then discussed how to change the design with custom cells, as well as how to implement editing features such as swipe-to-delete.