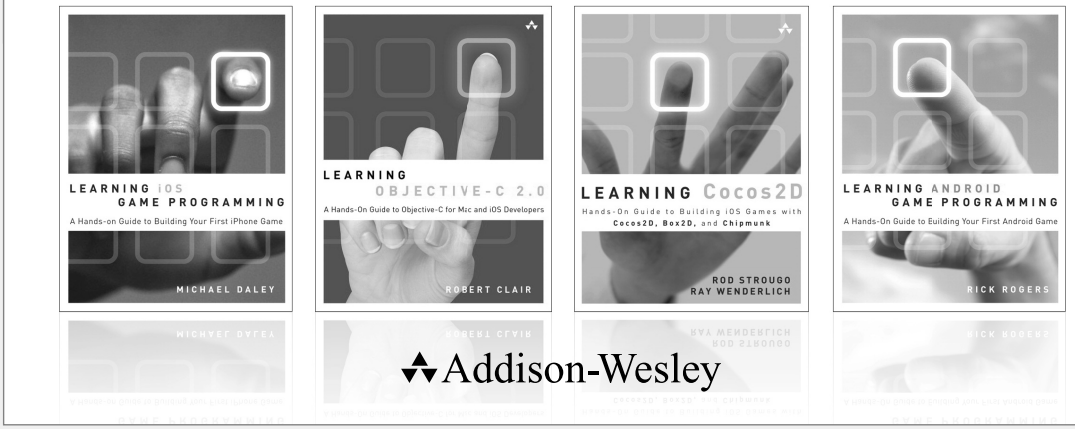
LEARNING MonoTouch

A Hands-On Guide to Building iPhone and iPad Applications with C# and .NET

MICHAEL BLUESTEIN

Learning MonoTouch

Addison-Wesley Learning Series



Visit **informit.com/learningseries** for a complete list of available publications.

The **Addison-Wesley Learning Series** is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning MonoTouch

A Hands-On Guide to Building iOS Applications with C# and .NET

Michael Bluestein

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Bluestein, Michael, 1970-
Learning MonoTouch : a hands-on guide to building iOS applications with C# and .NET / Michael Bluestein.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-71992-8 (pbk. : alk. paper)
ISBN-10: 0-321-71992-1 (pbk. : alk. paper) 1. iPad (Computer)—Programming. 2. Application software—Development. 3. iOS (Electronic resource) 4. C# (Computer program language) 5. Microsoft .NET. I. Title.
QA76.8.I863B626 2012
006.7'882—dc23

2011018222

ISBN-13: 978-0-321-71992-8
ISBN-10: 0-321-71992-1

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing July 2011

Editor-in-Chief
Mark Taub

Acquisitions Editor
Chuck Toporek

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Bart Reed

Indexer
Erika Millen

Proofreader
Williams Woods Publishing Services

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

❖

For Rose, Lilly, and Joshua

❖

Contents at a Glance

Table of Contents

Preface

I first learned of the iPhone work being done by the Mono team while attending Miguel de Icaza's Mono presentation at Microsoft's Professional Developer Conference in 2008. Miguel talked about how they were bringing .NET and C# development to the iPhone as part of the Unity3D game platform. I found it fascinating that they were able to achieve this. I was already working with the iPhone at the time with Xcode and Objective-C and, like many people, saw it as an amazing platform. Having spent many years working with .NET and C#, the idea of being able to use those technologies on the iPhone intrigued me. This would be the combination of two of my favorite technologies, .NET and the iPhone. However, my focus has never been on game development, so I didn't pursue it at the time.

The next year, Joseph Hill came to our local .NET code camp and gave a presentation on Mono in general. By this point, the Mono team had already started developing the product that would become MonoTouch. Joseph mentioned that a private beta would be coming up soon and to contact him if interested. I thought this would be great even if they could offer a fraction of what's available from Objective-C, or from .NET in general, because it would offer additional options for solving iPhone problems, such as garbage collection and perhaps a bit of code reuse.

Little did I know they would bring to the table everything I was able to do with Objective-C on the iPhone and most of .NET as well. Also, as it turned out, the MonoTouch team and community is full of knowledgeable people that are truly passionate about what they do. The story around MonoTouch is a powerful one because while increasing your toolset to solve problems on iOS, you don't sacrifice user-experience or platform capabilities. I've enjoyed every moment I've had working with MonoTouch and am sure you will as well.

The Audience for This Book

This book is primarily for .NET/C# developers with several years of application development experience, but little to no iPhone or Mac development experience. It assumes intermediate-level knowledge of C#/.NET. However, if you're an Objective-C developer, it covers many core iOS concepts that are language agnostic, so there's something here for you, too. This book teaches C#/.NET developers how to take their existing skills to the iPhone to build iOS applications using MonoTouch.

About the Sample Code

All of the code examples are available on my Github account at https://github.com/mikebluestein.

Acknowledgments

I'd like to first thank my wife and kids for putting up with the time I spent away from them to write this book. Without their encouragement, I wouldn't have been able to do this. As you can imagine, lots of people are involved in the creation of a book. Thanks to the team at Pearson for supporting this effort, including Chuck Toporek, Sheri Cain, Olivia Basegio, Bart Reed, and Anne Goebel, as well as all the people behind the scenes. I'd also like to thank everyone on the awesome Mono and MonoTouch team for creating such an amazing platform, particularly Joseph Hill for all the support and encouragement he has consistently offered from the moment I got involved in the MonoTouch community and Miguel de Icaza for the amazing work he does for developers everywhere. I'd especially like to thank Geoff Norton for leading the creation of MonoTouch and the unparalleled support and guidance he gives to everyone. I can't tell you how many times I've been up late at night stuck on something and he has been there to support me—and anyone else—in any way he can. Also, thanks to Geoff, Chris Hardy, and Robert Kozak for their technical review of the book. It has been a pleasure working on a book about MonoTouch, and I hope you enjoy reading it as much as I have writing it.

About the Author

Michael Bluestein has been working with MonoTouch since the first private beta release and is an active member of the MonoTouch community. His applications are among the first using MonoTouch to be published in Apple's App Store. A former Principal Software Engineer at Dassault Systèmes Solidworks Corporation, he has developed software professionally since the early 1990s. His blog on various development topics, including MonoTouch, can be found at mikebluestein.wordpress.com.

Introduction

Welcome to *Learning MonoTouch*. If you are a .NET developer interested in building native applications for iOS devices, MonoTouch is a great choice. It blends the Cocoa-Touch frameworks and Objective-C language concepts elegantly with C# and .NET, resulting in a very well designed technology that is a pleasure to work with. You can use MonoTouch for App Store deployment and enterprise deployment as well (assuming you have the appropriate license). There is even a simulator-only, free version, so you can get started learning and trying it out without any additional cost. Also, if you're a student, a discounted student edition is available.

MonoTouch allows you to create applications using the same APIs available in Objective-C, while at the same time offering many of the language and API features from Mono, C#, and .NET. In addition to nicely abstracted Objective-C memory management, you get garbage collection, reuse of non-UI code, ADO.NET wrappers on SQLite, web services, Linq, and generics—to name just a few things.

MonoTouch is great because it complements and builds upon the technologies from Apple while adding a plethora of extra functionality to help you in developing applications. The team and community around MonoTouch are also worth taking note of. You can participate in forums, a mailing list, and the very active IRC channel to get support from members of the MonoTouch team and the community, discuss your ideas, or just hang out. The community is one of the best things about MonoTouch. Jump on IRC and you will find everyone from new MonoTouch developers to the creators of Mono and MonoTouch themselves actively working to make developers' experiences great.

How This Book Is Organized

Learning MonoTouch has 12 instructional chapters to help you learn everything you need to know about using MonoTouch for iOS development; the chapters are described in the following sections.

Chapter 1: Hello MonoTouch

This chapter starts off with a walkthrough of how to get the development environment set up, along with a basic discussion of the various development tools used in Mono-Touch development. It then presents the development of a simple application, followed by an explanation of its internals. The chapter concludes with instructions on getting the app deployed on a device and debugging it with the MonoTouch soft-debugger.

Chapter 2: iOS SDK via MonoTouch

This chapter explains how MonoTouch abstracts the iOS SDK to allow development against native classes from C#. Starting off with an overview of the iOS SDK, a simple example is presented in Objective-C and then contrasted against its C# counterpart. Using this example, this chapter explains how to work with outlets and compares common iOS development patterns, showing how to use them from C#. The chapter concludes with an overview of memory management in Objective-C versus garbage collection in MonoTouch, showing how MonoTouch takes care of this for you and when you need to consider the Objective-C model from C# code.

Chapter 3: Views and View Controllers

This chapter shows how to structure a MonoTouch application for the Model-View-Controller (MVC) design pattern. It introduces the `UIView` and `UIViewController` classes and shows how to work with them in code as well as from Interface Builder by way of examples that demonstrate touch support and the accelerometer.

Chapter 4: Common iOS Classes

This chapter explains how to use several basic classes common in iOS development. It presents many of the views and controls that ship with the iOS SDK to aid in creating user interfaces, as well as several controllers that abstract various capabilities such as the address book, camera access, sending email, and playing music from the iPod library.

Chapter 5: Tables and Navigation

This chapter introduces the `UITableView` and `UITableViewController` and presents some common usage scenarios where tables are typically used. A discussion of the basic pattern for using `UITableViewController` is presented along with a few ways to customize the `UITableView` to provide a richer experience, both visually and in performance. This chapter also introduces `UINavigationController` and shows you how to use it in conjunction with `UITableViewController`.

Chapter 6: Graphics and Animation

This chapter discusses the graphics and animation subsystems—Core Graphics and Core Animation, respectively—and explains how they are used under UIKit to form the basis for much of what you see in iOS.

Chapter 7: Core Location

This chapter presents the Core Location framework and shows how you can use it directly to get location data using a variety of positioning technologies such as cell tower triangulation, Wi-Fi, and GPS. It then delves into some of the newer location technologies such as significant location changes and region monitoring.

Chapter 8: MapKit

This chapter discusses the MapKit framework, including the `MKMapViewControl`, and shows how to create interactive maps in your applications. It explains MapKit's integration with Core Location, along with how to add annotations and overlays to maps to build customized mapping experiences.

Chapter 9: Connecting to Web Services

This chapter shows how to consume web services from MonoTouch using several available technologies. It discusses how to consume SOAP-based web services, REST services, as well as JSON, XML, RSS, and WCF as they apply to MonoTouch development. In addition to .NET, the chapter also shows how to use the CocoaTouch HTTP stack from MonoTouch.

Chapter 10: Networking

This chapter presents the networking features offered by the GameKit framework for providing service discovery and networking over Bluetooth, including how to create voice communication between devices. It then shows how to use Bonjour directly to publish and discover services, along with using familiar .NET networking technologies such as TcpClient.

Chapter 11: Saving Application Data

This chapter shows how to use several of the data storage technologies available under iOS when using MonoTouch, such as the ADO.NET provider to SQLite, .NET serialization, and `NSUserDefaults`. It also begins a sample application that is used in the following chapter on iPad development as well.

Chapter 12: iPad Development

This chapter covers several of the classes offered specifically for developing iPad applications. It continues on the sample application from the previous chapter, demonstrating how to take an iPhone application and extend it into a universal application targeting the iPad, in addition to the iPhone and iPod Touch.

4

Common iOS Classes

iOS contains a number of controls and classes that help you considerably when building applications. The user interface elements range from the buttons and labels we have already seen, to sliders, progress views, and paging controls—to name just a few. There are also classes to abstract various system capabilities, such as playing music and sending email. In this chapter, we'll survey several of the more common classes you'll use when building applications.

User Interface Views and Controls

UIKit contains various `UIControl` subclasses. `UIControl` itself derives from `UIView` and adds a variety of events to deal with user interaction. An example of a control we've already seen is the `UIButton` class. UIKit additionally contains a number of other controls.

`UISegmentedControl`

The `UISegmentedControl` is basically a tabbed interface control within a view. It is typically used to allow users to specify a particular set of subviews to interact with, effectively grouping them together, although it could also be used to create a menu structure of sorts. The control is composed of several buttons, broken into segments. Each button can have a title and an image, and the control itself can take on multiple styles.

To create a `UISegmentedControl` and add segments to it, you simply pass the titles you want for each segment to the constructor. For example, here is one way to create a `UISegmentedControl` with four segments, with their respective titles set:

```
public partial class ControlDemoViewController : UIViewController
{
    UISegmentedControl _segmentedControl;
    ...
    public override void ViewDidLoad ()
    {
        base.ViewDidLoad ();

        _segmentedControl = new UISegmentedControl(new object[]{"one",
```

```
            "two", "three", "four"});
        _segmentedControl.Frame =
            new RectangleF (10, 10, View.Frame.Width - 20, 50);

        View.AddSubview(_segmentedControl);
    }
}
```

> **Note**
>
> In the examples here we have added a view controller named `ControlDemoViewController`.

This results in the default `UISegmentedControl` (`UISegmentedControlStyle.Plain`) shown in Figure 4.1.

Figure 4.1 Default `UISegmentedControl` with segment titles

Setting the `ControlStyle` to any of the values in the `UISegmentedControlStyle` enumeration can change the style of the control. As mentioned, you can use an image for each segment. Additionally, you can easily set the selected segment and tint color. Here is an example that sets a bezeled style with a black tint color and an image set on the first segment, which is also set as "selected" (see Figure 4.2):

```
_segmentedControl = new UISegmentedControl (new object[] { "one",
    "two", "three", "four" });
_segmentedControl.ControlStyle = UISegmentedControlStyle.Bezeled;
_segmentedControl.TintColor = UIColor.Black;
_segmentedControl.SetImage (UIImage.FromFile ("Star.png"), 0);
_segmentedControl.SelectedSegment = 0;
_segmentedControl.Frame = new RectangleF (10, 10,
    View.Frame.Width - 20, 50);
```

Figure 4.2 `UISegmented` control with additional customizations

You can also set images via the constructor in the same way you set titles. Additionally, you can set each title directly via the `SetTitle` method. You can even mix images and titles into the constructor because the array it takes is an array of objects. For example, the following will result in the same `UISegementedControl` shown in Figure 4.2:

```
_segmentedControl = new UISegmentedControl (new object[] {
    UIImage.FromFile ("Star.png"), "two", "three", "four" });
_segmentedControl.ControlStyle = UISegmentedControlStyle.Bezeled;
_segmentedControl.TintColor = UIColor.Black;
_segmentedControl.SelectedSegment = 0;
_segmentedControl.Frame = new RectangleF (10, 10,
    View.Frame.Width - 20, 50);
```

To handle the changes to the selected segment, you can register for the `ValueChanged` event. In the handler, you can do whatever you like based on the current selected segment, such as hiding certain views or changing some state. For example, here we simply added a `UILabel` as a subview of the `UISegmentedControl` and set its text as the selected segment changes (see Figure 4.3):

```
string _text;
UILabel _testLabel;

...

_testLabel = new UILabel(){Frame = new RectangleF(10, 200, 100, 50)};

_segmentedControl = new UISegmentedControl (new object[] {
    UIImage.FromFile ("Star.png"), "two", "three", "four" });
_segmentedControl.ControlStyle = UISegmentedControlStyle.Bezeled;
_segmentedControl.TintColor = UIColor.Black;
_segmentedControl.Frame = new RectangleF (10, 10,
    View.Frame.Width - 20, 50);

_segmentedControl.ValueChanged += (o, e) => {
    _selectedTitle = _segmentedControl.TitleAt
        (_segmentedControl.SelectedSegment) ?? "Title not set";
    _testLabel.Text = _text;
};

_segmentedControl.SelectedSegment = 0;
_segmentedControl.AddSubview(_testLabel);
```

Figure 4.3 Handling `UISegmentedControl ValueChanged`

UISlider

The `UISlider` control is similar to slider controls found on other platforms, except it allows you to move the slider via touch. You can initialize a slider with the minimum, maximum, and initial values. The default slider appearance is shown in Figure 4.4. Here, we are capturing the value as it is changed and assigning it to a label's text in the slider's `ValueChanged` event:

```
UISlider _slider;
...
_slider = new UISlider { Frame = new RectangleF (10, 10,
    View.Frame.Width - 20, 50) };
_slider.MinValue = 0.0f;
_slider.MaxValue = 20.0f;
_slider.SetValue (10.0f, false);
```

```
_slider.ValueChanged += delegate {
    _text = _slider.Value.ToString ();
    _testLabel.Text = _text;
};
```

Figure 4.4 `UISlider` default appearance

The slider can also be customized to set thumb images, track images, and min/max value images (see Figure 4.5).

```
_slider = new UISlider { Frame = new RectangleF (10, 10,
    View.Frame.Width - 20, 50) };
_slider.MinValue = 0.0f;
_slider.MaxValue = 20.0f;
_slider.SetValue (10.0f, false);

_slider.ValueChanged += delegate {
    _text = _slider.Value.ToString ();
```

```
    _testLabel.Text = _text;
};

// Customize the look and feel of the slider

_slider.SetThumbImage (UIImage.FromFile("Thumb0.png"),
    UIControlState.Normal);
_slider.SetThumbImage (UIImage.FromFile("Thumb1.png"),
    UIControlState.Highlighted);
_slider.SetMaxTrackImage (UIImage.FromFile("MaxTrack.png"),
    UIControlState.Normal);
_slider.SetMinTrackImage (UIImage.FromFile("MinTrack.png"),
    UIControlState.Normal);
_slider.MaxValueImage = UIImage.FromFile("Max.png");
_slider.MinValueImage = UIImage.FromFile("Min.png");
```

Figure 4.5 `UISlider` with customized look and feel

UISwitch

The `UISwitch` control is used to toggle between two states, such as on or off. It's basically designed to simulate a physical on/off switch. You'll find it commonly used in application settings. The control itself defaults to an off state, but you can set its value programmatically using the `SetState` method, with the first argument being the on/off value and the second whether the switch animates initially from off to on (if the initial value is set to on). To capture changes in the switch value, you handle the `ValueChanged` event. The switch's `On` property is a Boolean containing the on/off value, which will be true when on. Even though a `UISwitch` ultimately is a `UIView`, it is designed to always be the same size. You can set the position of its frame, but the size is ignored. Figure 4.6 shows a `UISwitch` where changes to its value are written to a `UILabel`.

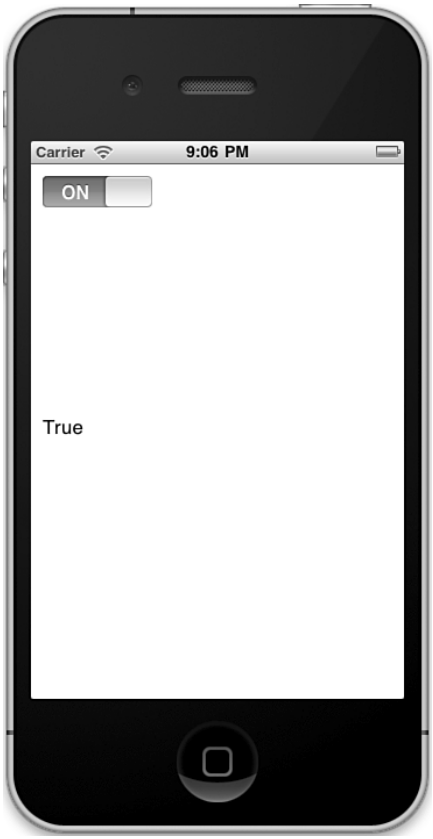


Figure 4.6 `UISwitch` value written to a `UILabel`

```
UISwitch _switch;
...
_switch = new UISwitch {Frame = new RectangleF (
    new PointF(10,10), SizeF.Empty)};
_switch.SetState (true, false);
_switch.ValueChanged += delegate {
    _text = _switch.On.ToString ();
    _testLabel.Text = _text;
};
```

`UIPageControl` and `UIScrollView`

The `UIPageControl` is used to designate which page you are on in an application where the pages typically slide horizontally across the screen. The actual paging can be implemented with a `UIScrollView`, where the `UIPageControl` tracks the current page via a series of dots. This is the experience you see on the home screen of an iOS device when moving across pages of applications.

The `UIScrollView` supports adding content that is too large to fit in a designated area of the screen and have it be scrollable either horizontally, vertically, or both. It is well suited for implementing a paged experience, where each view that acts logically as a page is added as a subview of the `UIScrollView`. The `UIScrollView` even includes a `PageEnabled` property that when set to true will cause the scrolling to snap to each page. To make scrolling happen, you set the `ContentSize` of the `UIScrollView` to something larger than the frame of the `UIScrollView`. Combined with setting `PageEnabled`, the `UIScrollView` will determine the physical page size internally. For any subviews you add to act as pages, you simply set their position offset appropriately within the content size of the scroll view. Listing 4.1 shows an example of adding basic views, each with a single label containing the page number, where sliding the views left or right will allow you to page between them.

Listing 4.1 **Paging with a `UIScrollView`**

```
public partial class PagingController : UIViewController
{
    UIScrollView _scroll;
    List<UIView> _pages;

    int _numPages = 4;
    float _padding = 10;
    float _pageHeight = 400;
    float _pageWidth = 300;

    ...

    public override void ViewDidLoad ()
    {
```

```
        base.ViewDidLoad ();

        View.BackgroundColor = UIColor.Black;

        _pages = new List<UIView> ();

        _scroll = new UIScrollView {
            Frame = View.Frame,
            PagingEnabled = true,
            ContentSize = new SizeF (
                _numPages * _pageWidth + _padding
                + 2 * _padding * (_numPages - 1),
                View.Frame.Height)
        };

        View.AddSubview (_scroll);

        for (int i = 0; i < _numPages; i++) {
            UIView v = new UIView ();
            v.Add( new UILabel{
                Frame = new RectangleF (100, 50, 100, 25),
                Text = String.Format ("Page {0}", i+1)}
            );

            _pages.Add (v);
            v.BackgroundColor = UIColor.Gray;

            v.Frame = new RectangleF (
                i * + _pageWidth + _padding + (2 * _padding * i),
                0, _pageWidth, _pageHeight);

            _scroll.AddSubview (v);
        }
    }
}
```

To keep track of the current page, you can use the `UIPageControl`. This control shows a series of dots, where the number of dots represents the page count and the current page number is denoted by the highlighted dot. The control itself is not physically connected to the scroll view, so it is up to you to add the code for the page count and to track the current page. The `UIPageControl`'s `Pages` property sets the page count. The formula for the current page in the scroll view is simply the current offset, available via the scroll view's `ContentOffset` property, divided by the page width. Setting this to the `UIPageControl`'s `CurrentPage` changes the highlighted dot to the proper page (see Figure 4.7).

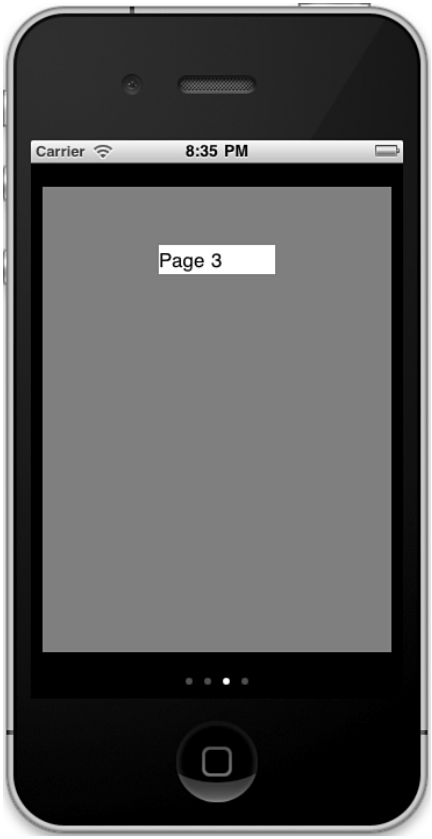


Figure 4.7 Paging with `UIScrollView` and `UIPageControl`

```
public partial class PagingController : UIViewController
{
    UIPageControl _pager;
    ...

    public override void ViewDidLoad ()
    {
        ...

        _scroll.Scrolled += delegate {

            _pager.CurrentPage =
                (int)Math.Round(_scroll.ContentOffset.X/_pageWidth);

        };
```

```
        _pager = new UIPageControl();
        _pager.Pages = _numPages;
        _pager.Frame = new RectangleF(0, 420, View.Frame.Width, 50);

        View.AddSubview(_pager);
    }
}
```

In addition to `UIControl` subclasses, UIKit has a number of classes that derive directly from `UIView`, such as the `UIScrollView`. Additional classes present rich information displays such as advertisements, web pages, maps, and tables. We'll cover some of these classes here. Others, such as `UIMapView` and `UITableView`, are covered in later chapters.

UIActivityIndicatorView

The `UIActivityIndicatorView` is used to indicate some operation is in progress in an indeterminate fashion. It presents itself as an animated rotating circle of sorts while the operation is happening. To use a `UIActivityIndicatorView`, you add it as a subview like any other view. To make it actually appear and start animating, you call its `StartAnimating` method. Likewise, to stop the animation and make the activity indicator disappear, you call `StopAnimating`. It's worth noting that any long-running operation you are performing would need to happen on a different thread; otherwise, you'll block the main thread and you would never see the activity indicator. Listing 4.2 demonstrates how to create a `UIActivityIndicatorView` to indicate an operation, implemented in the `DoSomething` method, is in progress. The result is shown in Figure 4.8.

Listing 4.2 **`UIActivityIndicatorView` Implementation**

```
...

UIActivityIndicatorView _activityView;

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

    showActivityButton.TouchUpInside +=
        HandleShowActivityButtonTouchUpInside;
}

void HandleShowActivityButtonTouchUpInside (object sender, EventArgs e)
{
    _activityView = new UIActivityIndicatorView ();

    _activityView.Frame = new RectangleF (0, 0, 50, 50);
    _activityView.Center = View.Center;

    _activityView.ActivityIndicatorViewStyle =
```

```
        UIActivityIndicatorViewStyle.WhiteLarge;
    View.AddSubview (_activityView);
    _activityView.StartAnimating ();

    Thread t = new Thread (DoSomething);
    t.Start ();
}

void DoSomething ()
{
    Thread.Sleep (3000);

    using (var pool = new NSAutoreleasePool ()) {
        this.InvokeOnMainThread (delegate {
            _activityView.StopAnimating (); });
    }
}
```

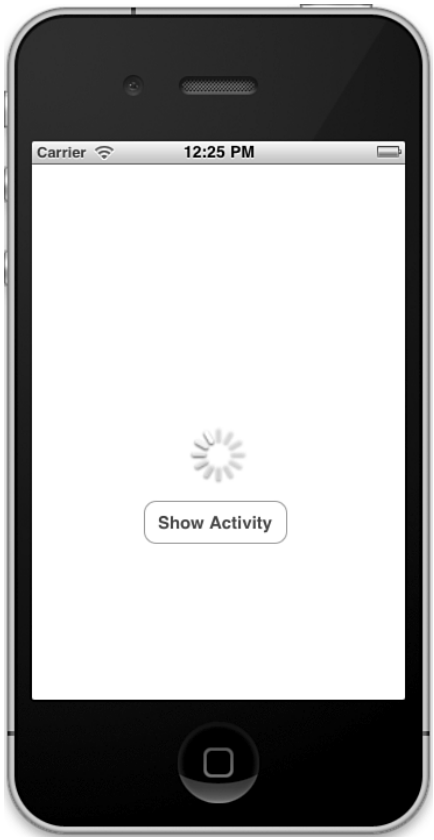


Figure 4.8 `UIActivityIndicatorView`

UIProgressView

Similar to the `UIActivityIndicatorView`, the `UIProgressView` is used to indicate some operation is underway. However, the `UIProgressView` is determinate because it displays the percentage of work that has been completed by filling in a portion of a horizontal bar. The progress is set using the `Progress` property and a floating-point value between 0 and 1, where 1 indicates 100% completion. Listing 4.3 implements a simulated operation whose progress is tracked with a `UIProgressView`, the result of which is shown in Figure 4.9.

Listing 4.3 **Example of a `UIProgressView`**

```
...

UIProgressView _progressView;

void HandleShowActivityButtonTouchUpInside (object sender, EventArgs e)
{
    _progressView = new UIProgressView ();
    _progressView.Frame = new RectangleF (0, 0, View.Frame.Width - 20,
        100);
    _progressView.Center = View.Center;
    _progressView.Style = UIProgressViewStyle.Default;

    View.AddSubview (_progressView);

    Thread t = new Thread (DoSomethingElse);
    t.Start ();
}

void DoSomethingElse ()
{
    int n = 3;

    for (int i = 0; i < n; i++) {
        Thread.Sleep (1000);

        using (var pool = new NSAutoreleasePool ()) {

            this.InvokeOnMainThread (delegate {
                _progressView.Progress = (float)(i + 1) / n; });
        }
    }
}
```

Figure 4.9 `UIProgressView`

`UIImageView`

We used `UIImageView` previously in Chapter 2, "iOS SDK via MonoTouch." You'll recall its purpose in life is to present a `UIImage` on the screen. The simplest example of a `UIImageView` sets the `Image` property and adds the view to the screen via a subview. For example, assuming the project has a file named monkey.png with a build action of Content, you would fill the screen with the image (see Figure 4.10) like this:

```
UIImageView _imageView;
...
_imageView = new UIImageView ();
_imageView.Frame = new RectangleF(0,0,
    View.Frame.Width, View.Frame.Height);
_imageView.Image = UIImage.FromFile("monkey.png");
```

Figure 4.10 `UIImageView` displaying a `UIImage`

To control how the view lays out its contents (for example, to preserve the image's aspect ratio), you use the `ContentMode` property:

```
_imageView = new UIImageView ();
_imageView.Frame = new RectangleF(0,0,
    View.Frame.Width, View.Frame.Height);
_imageView.Image = UIImage.FromFile("monkey.png");
 imageView.ContentMode = UIViewContentMode.ScaleAspectFit;
```

Here, we set the `ContentMode` to `ScaleAspectFit`, resulting in the image layout shown in Figure 4.11. Additionally, you can experiment with several other settings for `ContentMode` to control the layout to your liking.

Figure 4.11 `UIImageView` with `ContentMode` set to `ScaleAspectFit`

`UIWebView`

`UIWebView` is a wrapper around WebKit that you can use in your applications. You can use it to render HTML content either from the Internet or from a local resource. To demonstrate how to use it, let's build a simple browser application. Create a new window-based application and add a new view controller with a view. I named mine **LMT4-5** and **SimpleBrowserController**, respectively. After going through the steps to load the `SimpleBrowserController`'s view when the app finishes launching as usual, open the `SimpleBrowserViewController` in Interface Builder, where we'll do some of the work for this example.

For this example of a web browser, we'll support back, forward, and refresh functionality. We'll allow URL entry using a `UITextField` with a keyboard type of URL and a return key of "Go," which we can set in the Text Input Traits section of IB. Also, we'll nest all the navigation controls in a `UIToolbar`. Figure 4.12 shows the final setup of everything in IB, including the required outlet connections.

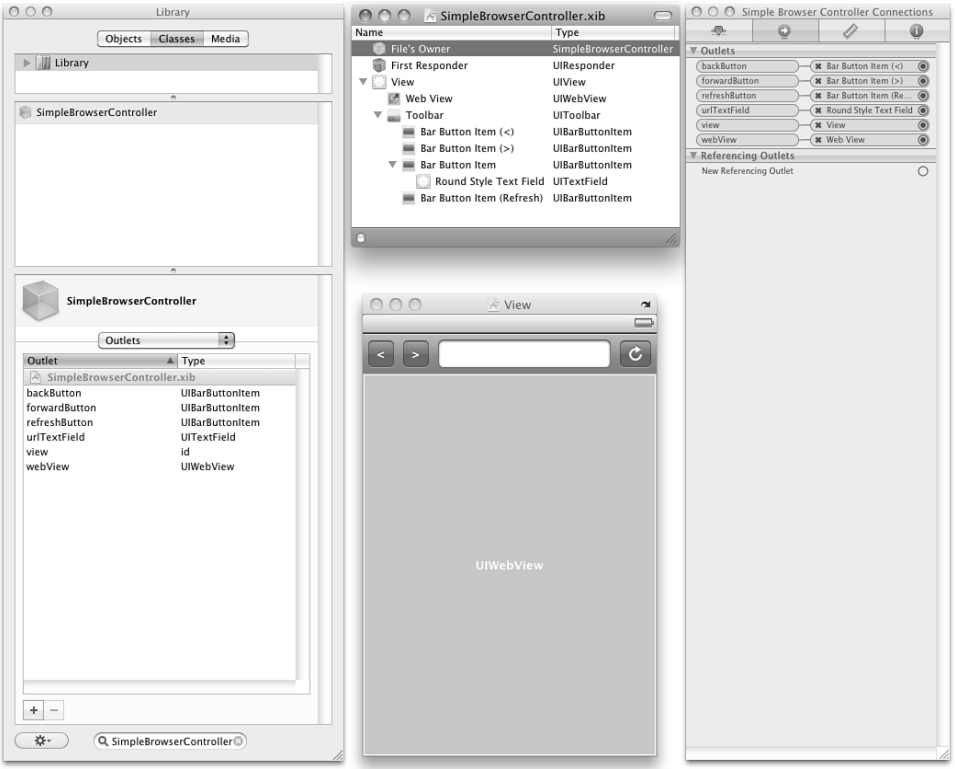


Figure 4.12 `SimpleBrowserController`'s view in Interface Builder

We want the URL entered in the `UITextField` to result in navigation to the web page when the user selects Go on the keyboard. To achieve this we use `ShouldReturn` on the `UITextField`. The function we assign to `ShouldReturn` takes care of building the `NSUrlRequest` from the URL entered by the user. The `UIWebview`'s `LoadRequest` method takes an `NSUrlRequest`, which it uses to load the web page. Also, to make the keyboard disappear, we call `ResignFirstResponder` on the `UITextField`.

```
urlTextField.ShouldReturn = textField =>
{
    textField.ResignFirstResponder ();
    string url = textField.Text;
    if (!url.StartsWith ("http"))
        url = String.Format ("http://{0}", url);
    NSUrl nsurl = new NSUrl (url);
    NSUrlRequest req = new NSUrlRequest (nsurl);
    webView.LoadRequest (req);
    return true;
};
```

In addition to `LoadRequest`, the `UIWebView` comes with other features to control navigation, such as moving back and forward through the page history and reloading the current page. These are implemented with the methods `GoBack`, `GoForward`, and `Reload`, respectively. In our example, we wire these up to the appropriate buttons:

```
backButton.Clicked += delegate { webView.GoBack (); };
forwardButton.Clicked += delegate { webView.GoForward (); };
refreshButton.Clicked += delegate { webView.Reload ();
```

Putting these snippets together in the `SimpleBrowserController`'s `ViewDidLoad` implementation gives us a simple running browser application where we can navigate to web pages and scroll them. However, our implementation does not support pinch zooming and does not fit the page to the screen by default. To include zoom support and fit-the-page functionality, simply set the `ScalesPageToFit` property to true. The resulting application is shown in Figure 4.13.

Figure 4.13 Simple browser application

ADBannerView

With iOS 4, Apple introduced the iAds program, which allows developers to easily include advertisements in their applications. The technology you use to include them is the `ADBannerView`, found in the `MonoTouch.iAd` namespace. iAds are supported on any device running iOS 4.x. Initially, this meant only iPhone and iPod Touch, but as of iOS 4.2, the iPad is also included.

For this example, we'll build a universal application for both the iPhone and iPad. Universal apps include support for native UI optimized for both the iPhone and iPad in a single executable, as opposed to the pixel doubling you'll get on the iPad if you only target the iPhone. We'll discuss this at length later in the book. For now, we're just making a universal app to demonstrate iAds on both devices.

Create a new project named LMT4-6 using the Universal Window-based Project template. Note that two files are added to this project. The MainWindowIPad.xib file is used when running the app on an iPad and the MainWindowIPhone.xib file is used on an iPhone. Along with these files are separate `AppDelegate` files for each target.

> **Tip**
>
> You can switch between the iPhone and iPad simulator for the project by setting the iPhone Simulator Target option under MonoDevelop's Project menu (see Figure 4.14).

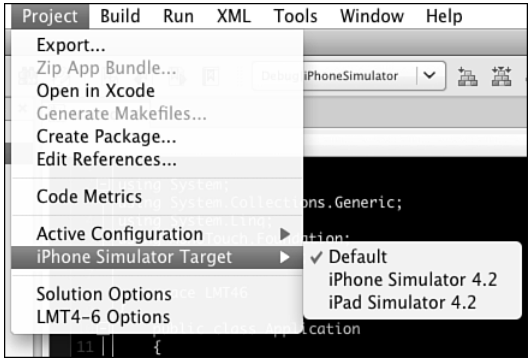


Figure 4.14 Changing the simulator target in MonoDevelop

Working with the `ADBannerView` is just like working with any other view. For callbacks, you can use the `ADBannerViewDelegate` or the .NET events that abstract it. Because this is a universal app, you'll need to add an iPad view with a controller in addition to an iPhone view with a controller. The process for adding the controller's view to the screen is the same as in previous examples, where you added the controller to MainWindow.xib, set the class and xib name, connected an outlet to the controller from the

AppDelegate, and loaded the controller's view in FinishedLaunching. The only difference here is you'll have to do it for both the iPad version of the various classes as well as the iPhone version.

For the view controllers, use the names DemoADIPad and DemoADIPhone, respectively. Using IB, add an ADBannerView to the views for each, associating connections named adBanner in both cases. Without any additional code, this would result in a test ad loading. However, there are a couple things you need to handle.

> **Note**
>
> When you drop an ADBannerView onto the design surface in IB, if you get an error, denoted by a red error icon in the lower right of the xib window, click the icon and set the deployment target version to 4.2.

First, in the event an ad doesn't load, you need to take care of hiding the banner. Subsequently, any further successful loading of ads needs to ensure the banner is visible. You can handle these situations using the FailedToReceiveAd and AdLoaded events, respectively.

Second, if you want to support multiple orientations in your applications, you will need to size the adBanner appropriately. You never should size it directly, though. You should size it implicitly by setting the CurrentContentSizeIdentifier. Also, the orientations you need are specified via the RequireSizeIdentifiers property, which defaults to both landscape and portrait. This property controls what banner view images are actually downloaded. For example, if you don't support multiple orientations in your application, you can tune the ad content to only download ad images for what you need.

To support rotating your view to both landscape and portrait orientations here, we simply override ShouldRotateToInterfaceOrientation, returning true. To introduce the aforementioned code to set the ADBannerView's currentContentSizeidentifier, you override WillRotate, setting the value appropriately for the orientation that is about to be rotated to. Listing 4.4 shows the controller implementation for the iPhone, with the iPad's implementation being identical. The resulting app, running on both the iPad and iPhone simulators, is shown in Figure 4.15, along with the test ad you'll get when touching the ADBannerView.

> **Tip**
>
> When submitting applications containing iAds to the App Store, don't include the test ad in your screenshots.

Listing 4.4 **View Controller Implementing iAD Support**

```
public partial class DemoADIPhone : UIViewController
{
    ...
    public override void ViewDidLoad ()
    {
        base.ViewDidLoad ();
```

```
        adBanner.AdLoaded += (s, e) => {
            Console.WriteLine ("Ad Loaded");
            ((ADBannerView)s).Hidden = false;
        };

        adBanner.FailedToReceiveAd += delegate(object sender,
            AdErrorEventArgs e)
        {
            Console.WriteLine("Ad failed to load. Error code = {0}",
                e.Error.Code);
            ((ADBannerView)sender).Hidden = true;
        };
    }

    public override void WillRotate (
        UIInterfaceOrientation toInterfaceOrientation, double duration)
    {
        base.WillRotate (toInterfaceOrientation, duration);
        if ((toInterfaceOrientation ==
                UIInterfaceOrientation.LandscapeLeft) ||
           (toInterfaceOrientation ==
                UIInterfaceOrientation.LandscapeRight))
        {
            adBanner.CurrentContentSizeIdentifier =
                ADBannerView.SizeIdentifierLandscape;
        }
        else
        {
            adBanner.CurrentContentSizeIdentifier =
                ADBannerView.SizeIdentifierPortrait;
        }
    }

    public override bool ShouldAutorotateToInterfaceOrientation
        (UIInterfaceOrientation toInterfaceOrientation)
    {
        return true;
    }
}
```

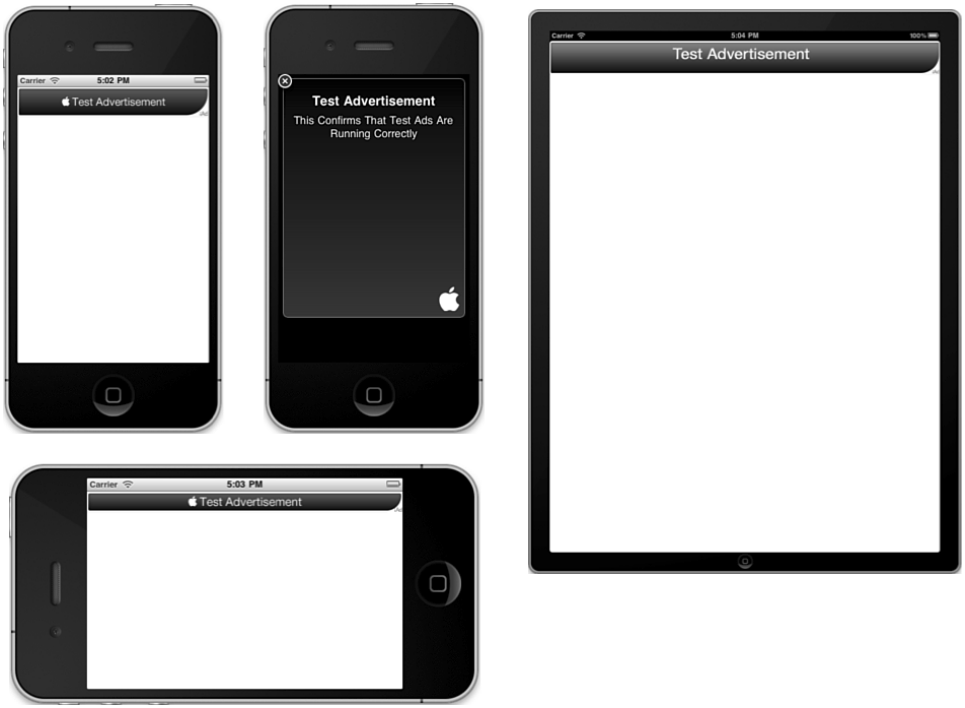


Figure 4.15 `ADBannerView` along with a test ad display

Device Capabilities

iOS devices have various capabilities. On the iPhone, for example, you can do everything from taking pictures and video with the camera to sending email and playing music. Much of the system is made available to integrate in your applications through a variety of built-in controllers.

`MFMailComposeViewController`

iOS includes built-in support for sending email from within your applications using the `MFMailComposeViewController`, found in the `MonoTouch.MessageUI` namespace. Simply check if the device is able to send mail using the `CanSendMail` property, and if it is true, bring up an `MFMailComposeViewController`. The controller has properties to add attachments (using the `AddAttachmentData` method), set the message body, send HTML mail, add CC recipients, and so on. After hydrating the `MFMailComposeViewController` and presenting its view, you can listen for completion results by subscribing to the `Finished` event or by overriding the `Finished` virtual function of `MFMailComposeViewController Delegate`. In the callback, you get back a result object, an error object, and the controller

itself via the `MFComposeResultEventArgs`, which you can use to present the completion status and dismiss the controller. Figure 4.16 shows a simple example of sending a string in the email message's body, using the code from Listing 4.5.

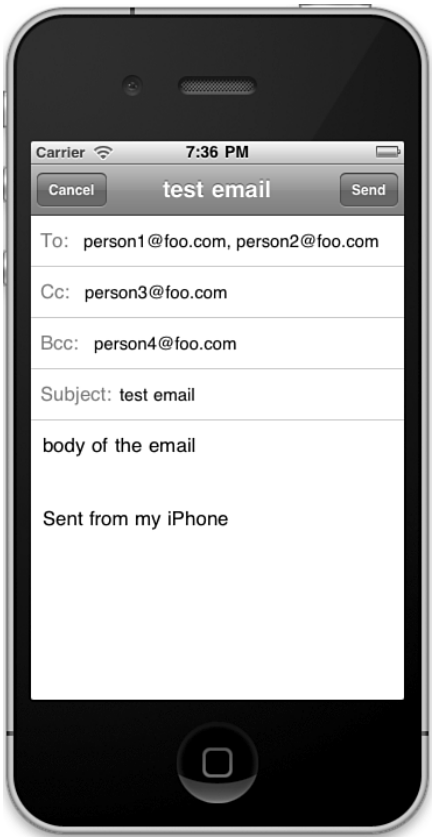


Figure 4.16 Sending email with `MFMailComposeViewController`

Listing 4.5 **`MFMailComposeViewController` Example**

```
MFMailComposeViewController _mail;
...

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

    mailButton.TouchUpInside += (o, e) =>
```

```
    {
        if (MFMailComposeViewController.CanSendMail) {
            _mail = new MFMailComposeViewController ();
            _mail.SetToRecipients (new string[] { "person1@foo.com",
                "person2@foo.com" });
            _mail.SetCcRecipients (new string[] { "person3@foo.com" });
            _mail.SetBccRecipients (new string[] { "person4@foo.com" })
            _mail.SetMessageBody ("body of the email", false);
            _mail.SetSubject ("test email");
            _mail.Finished += HandleMailFinished;
            this.PresentModalViewController(_mail, true);

        } else {
            var alert = new UIAlertView("Mail Alert",
                "Mail Not Sent", null, "Mail Demo", null);
            alert.Show();
        }
    };
}

void HandleMailFinished (object sender, MFComposeResultEventArgs e)
{
    if (e.Result == MFMailComposeResult.Sent)
    {
        var alert = new UIAlertView("Mail Alert", "Mail Sent",
            null, "Mail Demo", null);
        alert.Show();
    }
    e.Controller.DismissModalViewControllerAnimated(true);
}
```

`MPMediaPickerController` and `MPMusicPlayerController`

To select and play audio from your iPod library, you can use the `MPMediaPickerController` and `MPMusicPlayerController`, respectively. The `MPMediaPickerController` presents a view of your iPod library using a system view like the iPod application, but from within your application. You can use it to query and select items from your collection.

To determine the items that a user selects, you implement the `MPMediaPickerControllerDelegate`. This delegate's `MediaItemsPicked` method receives the items the user selects, as an `MPMediaItemCollection`. An `MPMediaItem` encapsulates metadata about the media item. For example, it contains artist and title information, among other things. You can use this to display the item's metadata in your app.

To play music, you can use the `MPMusicPlayerController`. This controller maintains a queue of items to play. Therefore, to play an item with it, you simply add items to the queue and subsequently call the `Play` method. The `MPMusicPlayerController` has a `SetQueue` method that you can use to pass along the `MPMediaItemCollection` sent to the `MPMediaPickerControllerDelegate`. There are also methods to control playback by performing actions such as pausing, stopping, and adjusting volume, making it easy to implement audio player functionality from within your applications.

Let's build a simple music player to demonstrate. After creating a new window-based iPhone application, add a view controller with a view named `MusicDemoController` and wire it up such that the view loads at startup, as usual. For this example, we'll support opening the iPod library for song selection, playback, stopping and pausing, as well as volume control. Also, we'll include labels to display the song's artist and title. Figure 4.17 shows the application's user interface in IB along with the various connections.

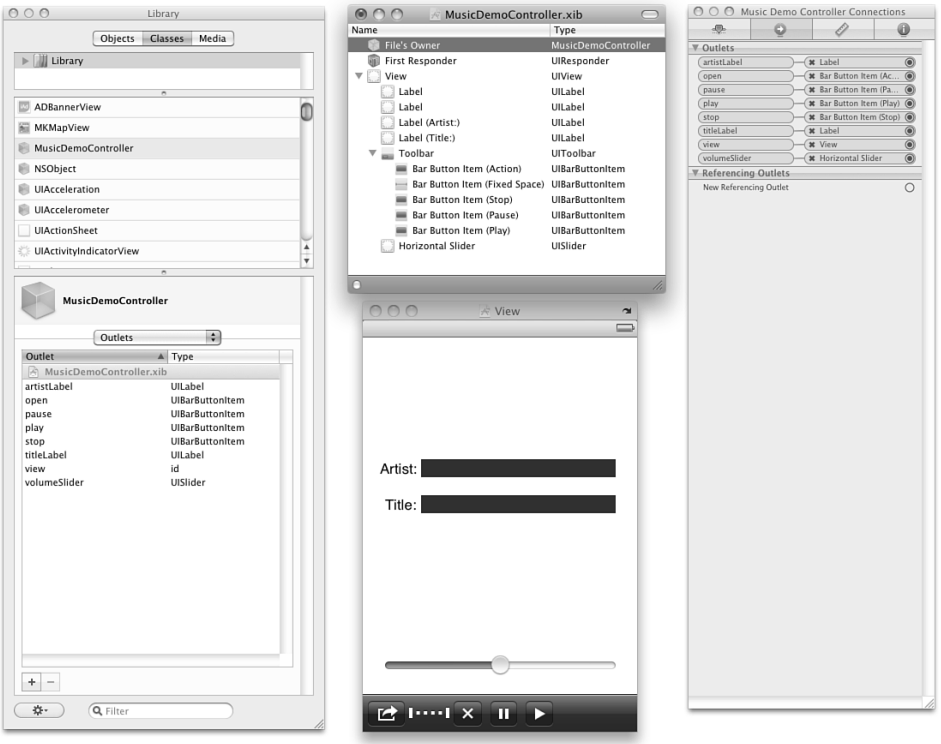


Figure 4.17 `MusicDemoController` view in Interface Builder

We use a slider for the volume control and add the various buttons to a toolbar, which is available via the `UIToolbar` class. For the bar button items, we get stock icon support by setting the appropriate identifier in the Attribute Inspector. When the user selects the

open button, denoted by the Action identifier (the one furthest to the left in Figure 4.17), we open the `MPMediaPickerController`. After selecting a song, we close the picker and populate the labels with the artist and title metadata.

> **Note**
> This example only works on a device.

At this point, we will have queued up the song in the `MPMusicPlayerController`, so we can play, pause, and stop it, along with change the volume. Listing 4.6 shows the implementation of the `MusicDemoController` to make all this happen.

Listing 4.6 **`MusicDemoController` Implementation**

```
public partial class MusicDemoController : UIViewController
{
    // Constructors omitted for brevity ...

    MPMusicPlayerController _musicPlayer;
    MPMediaPickerController _mediaController;
    MediaPickerDelegate _mpDelegate;

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

        _musicPlayer = new MPMusicPlayerController ();
        _musicPlayer.Volume = volumeSlider.Value;
        _mediaController = new MPMediaPickerController
            (MPMediaType.MPMediaTypeMusic);
        _mediaController.AllowsPickingMultipleItems = false;
        _mpDelegate = new MediaPickerDelegate (this);
        _mediaController.Delegate = _mpDelegate;

        volumeSlider.ValueChanged += delegate {
            _musicPlayer.Volume = volumeSlider.Value; };

        open.Clicked += (o, e) => {
            this.PresentModalViewController(_mediaController, true); };

        play.Clicked += (o, e) => { _musicPlayer.Play (); };

        pause.Clicked += (o, e) => { _musicPlayer.Pause (); };

        stop.Clicked += (o, e) => { _musicPlayer.Stop (); };
    }

    public class MediaPickerDelegate : MPMediaPickerControllerDelegate
    {
```

```
        MusicDemoController _viewController;

        public MediaPickerDelegate (
            MusicDemoController viewController) : base()
        {
            _viewController = viewController;
        }

        public override void MediaItemsPicked (MPMediaPickerController
            sender, MPMediaItemCollection mediaItemCollection)
        {
            _viewController._musicPlayer.SetQueue
                (mediaItemCollection);
            _viewController.DismissModalViewControllerAnimated (true);

            MPMediaItem mediaItem = mediaItemCollection.Items[0];

            //See MPMediaItem.h for various string property names
            //(Search for MPMediaItem.h in Mac Spotlight)

            string artist =
                mediaItem.ValueForProperty ("artist").ToString ();
            string title =
                mediaItem.ValueForProperty ("title").ToString ();

            _viewController.artistLabel.Text = artist;
            _viewController.titleLabel.Text = title;
        }

        public override void MediaPickerDidCancel
            (MPMediaPickerController sender)
        {
            _viewController.DismissModalViewControllerAnimated (true);
        }
    }
}
```

> **Note**
>
> Future versions of MonoTouch will expose the real `NSString` fields, such as `MPMediaItem.ArtistProperty`, so you won't have to search in MPMediaItem.h.

Address Book

iOS also includes support for interacting with the data you store in your system address book. This is the data you commonly access in the Phone, Contacts, and Mail applications. Much like the interaction you are afforded with iPod data, you have similar access to the address book from within your applications.

The address book is modeled in the `ABAddressBook` class. Using this class, you can enumerate objects that represent the people in your contact list and their associated data, such as phone numbers and email addresses, as shown here:

```
ABAddressBook ab = new ABAddressBook ();
ABPerson[] people = ab.GetPeople ();

foreach (ABPerson person in people) {

    Console.WriteLine("{0} {1}", person.FirstName, person.LastName);

    var phones = person.GetPhones ();

    if (phones.Count > 0) {

        foreach(var phone in phones)
            Console.WriteLine (" {0}, {1}", phone.Label, phone.Value);
   }
}
```

In addition to taking the approach of using `ABAddressBook` directly, you can use the `ABPeoplePickerNavigationController` to interact with the address book via the stock user interface. You can either use it simply to select a contact from the address book or navigate to contact details. To handle contact selection, you register for the `SelectPerson` event. From within your event handler, you can choose to allow navigation to contact details by setting the `Continue` property of the `ABPeoplePickerSelectPersonEventArgs` argument that is passed to the event handler. Setting `Continue` to true causes person selection to navigate to the contact details. The default setting is false, which you would use to simply pick the contact and subsequently dismiss the controller. The `event` argument also has a `Person` property, which is an `ABPerson`. Therefore, you can use the `Person` property to retrieve additional information, such as phones, as we did earlier. Listing 4.7 shows an example of using the `ABPeoplePickerNavigationController` to select a contact and add some of the person's information to a view, as well as dialing the contact's number.

Listing 4.7 **Using `ABPeoplePickerNavigationController`**

```
...
ABPeoplePickerNavigationController _peoplePicker;
ABPerson _person;
string _phoneNumber;

public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    _peoplePicker = new ABPeoplePickerNavigationController ();

    showPeoplePicker.TouchUpInside += delegate {
```

```
            this.PresentModalViewController (_peoplePicker, true); };

        _peoplePicker.Cancelled += delegate {
            this.DismissModalViewControllerAnimated (true); };

        _peoplePicker.SelectPerson += delegate(object sender,
            ABPeoplePickerSelectPersonEventArgs e) {

            // Setting Continue to true would allow navigation to the
            // contact's details, in which case you wouldn't dismiss the
            // controller below.
            //
            //e.Continue = true;

            _person = e.Person;

            nameLabel.Text = String.Format ("{0} {1}", _person.FirstName,
                _person.LastName);

            var phones = _person.GetPhones ();

            if (phones.Count > 0) {
                //just using the first phone for demo
                _phoneNumber = phones[0].Value;
                phoneLabel.Text = _phoneNumber;
            } else {
                _phoneNumber = String.Empty;
            }

            this.DismissModalViewControllerAnimated (true);
        };

        callPerson.TouchUpInside += delegate {

            if (!String.IsNullOrEmpty (_phoneNumber)) {

                NSUrl phoneUrl = new NSUrl (String.Format ("tel:{0}",
                    EscapePhoneNumber (_phoneNumber)));

                if (UIApplication.SharedApplication.CanOpenUrl (phoneUrl))
                    UIApplication.SharedApplication.OpenUrl (phoneUrl);
            }
        };
    }

    string EscapePhoneNumber (string phoneNum)
    {
```

```
    return phoneNum.Replace (" ", "-").Replace ("(", "")
        .Replace (")", "");
}
```

Here, we create a new `ABPeoplePickerNavigation` controller and display its view modally in response to the `TouchUpInsideEvent` of a `UIButton` named `showPeoplePicker`. When the user selects a person, we handle the selection in the `SelectPerson` event handler, populating labels with the person's name and the first phone number from the address book, after which we dismiss the controller. When the user touches another button named `callPerson`, we create an `NSUrl` using the system URL scheme for a phone number. Passing an `NSUrl` with this scheme to the `OpenUrl` method causes the phone application to launch and dial the number.

> **Note**
>
> In addition to the telephone, there are other URL schemes in iOS for things such as SMS text, iTunes, and maps—to name a few. See the *Apple URL Scheme Reference* for more details.

UIImagePickerController

The `UIImagePickerController` supports selecting images and videos from files stored on the device's photo library and albums, as well as capturing images and videos directly from the camera (for devices capable of image or video capture). The controller's view adapts to a stock user interface for library selection or camera interaction based on which scenario you ask for and which media types are available on the device. Subsequent callbacks are sent to the `UIImagePickerControllerDelegate`, where you can use the image or video in your application.

> **Note**
>
> Camera and video support varies based on the actual device. Newer-generation devices such as the iPhone 4 and the latest version of the iPod Touch support both.

The process of selecting or capturing images and videos is similar when using the `UIImagePickerController`. You simply create the controller and set the source and media types. The source type distinguishes between selecting media from the device and capturing it with the camera. The media type determines whether you are selecting or capturing images, video, or both. When you are selecting media from the device photo library, the media types you set will cause the content list to filter images and videos appropriately. Likewise, when you are capturing from the camera, the media types will direct the camera view into video or photo capture mode, or present a toggle button to switch between them if you set both image and video media types.

Let's look at an example. Here we will open an action sheet to allow the user to choose between either selecting media from the library or capturing it with the camera. Upon selection or capture of a photo, we will close the `UIImagePickerController` and

show the resulting image in a `UIImageView`. For video, we'll display a preview image of the first frame and provide a button to launch video playback using an `MPMediaPlayerController`. Figure 4.18 shows the setup in IB, with the implementation in Listing 4.8.

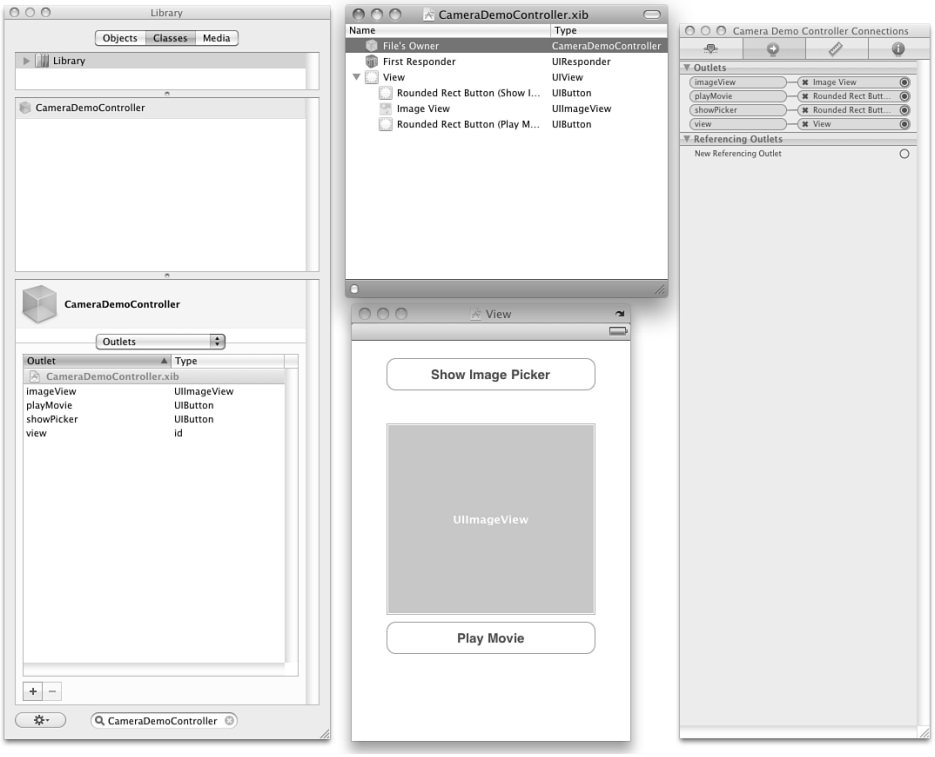


Figure 4.18 `CameraDemoController` in Interface Builder

Listing 4.8 **Using a `UIImagePickerController` for Photos or Video**

```
public partial class CameraDemoController : UIViewController
{
    UIImagePickerController _picker;
    PickerDelegate _pickerDel;
    UIActionSheet _actionSheet;
    MPMoviePlayerController _mp;
    // constructors ...

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

```
    _picker = new UIImagePickerController ();
    _pickerDel = new PickerDelegate (this);
    _picker.Delegate = _pickerDel;

    _actionSheet = new UIActionSheet ();
    _actionSheet.AddButton ("Library");
    _actionSheet.AddButton ("Camera");
    _actionSheet.AddButton ("Cancel");
    _actionSheet.CancelButtonIndex = 2;
    _actionSheet.Delegate = new ActionSheetDelegate (this);

    showPicker.TouchUpInside += delegate {
        _actionSheet.ShowInView (this.View); };

    playMovie.Hidden = true;

    playMovie.TouchUpInside += delegate {
        if (_mp != null) {
            View.AddSubview (_mp.View);
            _mp.SetFullscreen (true, true);
            _mp.Play ();
        }
    };
}

class ActionSheetDelegate : UIActionSheetDelegate
{
    CameraDemoController _controller;

    public ActionSheetDelegate (CameraDemoController controller)
    {
        _controller = controller;
    }

    void ShowPicker (UIImagePickerControllerSourceType sourceType)
    {
        if (!UIImagePickerController
            .IsSourceTypeAvailable (sourceType)) {

            var alert = new UIAlertView ("Image Picker",
                "Source type not available", null, "Close");
            alert.Show ();

        } else {

            _controller._picker.SourceType = sourceType;
```

```
            string[] availableMediaTypes = UIImagePickerController
                .AvailableMediaTypes (sourceType);
            string[] requestedMediaTypes = new string[] {
                "public.image", "public.movie" };
            List<string> mediaTypes = new List<string> ();

            foreach (string mediaType in requestedMediaTypes) {
                if (availableMediaTypes.Contains (mediaType))
                    mediaTypes.Add (mediaType);
            }

            _controller._picker.MediaTypes = mediaTypes.ToArray ();

            _controller.PresentModalViewController
                (_controller._picker, true);
        }
    }

    public override void Clicked (UIActionSheet actionSheet,
        int buttonIndex)
    {
        switch (buttonIndex) {
        case 0:
            ShowPicker (UIImagePickerControllerSourceType
                .PhotoLibrary);
            break;
        case 1:
            ShowPicker (UIImagePickerControllerSourceType
                .Camera);
            break;
        }
        actionSheet.DismissWithClickedButtonIndex (buttonIndex,
            true);
    }
}

class PickerDelegate : UIImagePickerControllerDelegate
{
    CameraDemoController _controller;

    public PickerDelegate (CameraDemoController controller)
    {
        _controller = controller;
    }

    public override void FinishedPickingMedia
```

```
            (UIImagePickerController picker, NSDictionary info)
        {
            picker.DismissModalViewControllerAnimated (true);

            string mediaType = info[new NSString
                ("UIImagePickerControllerMediaType")].ToString ();
            UIImage img = null;

            if (mediaType == "public.image") {

                img = (UIImage)info[new NSString
                    ("UIImagePickerControllerOriginalImage")];
                _controller.playMovie.Hidden = true;

            } else if (mediaType == "public.movie") {

                NSUrl videoUrl = (NSUrl)info[new NSString
                    ("UIImagePickerControllerMediaURL")];
                _controller._mp =
                    new MPMoviePlayerController (videoUrl);
                img = _controller._mp.ThumbnailImageAt (0,
                    MPMovieTimeOption.NearestKeyFrame);
                _controller.playMovie.Hidden = false;
            }

            if (img != null)
                _controller.imageView.Image = img;
        }
    }
}
```

For the action sheet, we add buttons to either choose the library or camera, or to cancel, handling the selection in the action sheet's delegate. Because capabilities vary between devices, you need to check that a particular source type is available using `UIImagePickerController`'s `IsSourceTypeAvailable` method. Once you know a source type is available, you can pass in available media types by setting them in a string array assigned to the `UIImagePickerController`'s `MediaTypes` property. Once this is done, with the `UIImagePickerController`'s delegate having been previously assigned, you can present the `UIImagePickerController`.

The view displayed by the `UIImagePickerController` varies based on the aforementioned settings. Once you select or capture a video or image, the resulting media can be harvested in the `UIImagePickerControllerDelegate`'s `FinishedPickingMedia` method. The media type is available in the `NSDictionary` that is passed into this method via the `UIImagePickerControllerMediaType` key. You can use this to run the appropriate code for either image or video post-processing.

Here, we simply display an image in an `ImageView` or, for video, display a preview image. Notice the preview image can be extracted from the video using the `Thumbnail ImageAt` method of an `MPMoviePlayerController`, which we can also use to play the video after it is selected.

Summary

iOS contains a plethora of classes that make it easier to create applications. In this chapter, we looked at some of the more common classes that provide user interface capabilities, as well as more intricate view controllers that abstract device capabilities, such as the camera. As you've seen, most classes in iOS follow similar design patterns, making learning additional features more approachable. Although in some cases these classes may be a bit more involved to work with, you'll find the basic design principles typically still apply. In the next chapter, we'll cover two such classes used in many iOS application scenarios: `UITableViewController` and `UINavigationController`.

A

B

C

D

E

F

G

H

I-J-K

L

M

N

O

P-Q

R

S

T

U

V

W

X-Y-Z