



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Xamarin Mobile Application Development for iOS

If you know C# and have an iOS device, learn to use one language for multiple devices with Xamarin

Paul F. Johnson

www.allitebooks.com

[PACKT]
PUBLISHING

Xamarin Mobile Application Development for iOS

If you know C# and have an iOS device, learn to use
one language for multiple devices with Xamarin

Paul F. Johnson

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Xamarin Mobile Application Development for iOS

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: October 2013

Production Reference: 1181013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78355-918-3

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author

Paul F. Johnson

Reviewers

Yaroslav Bigus

John Goodwin

Andri Yadi

Acquisition Editor

Kevin Colaco

Commissioning Editor

Shaon Basu

Copy Editors

Alisha Aranha

Dipti Kapadia

Kirti Pai

Lavina Pereira

Laxmi Subramanian

Technical Editors

Nikhita K. Gaikwad

Menza Mathew

Project Coordinator

Akash Poojary

Proofreader

Lawrence A. Herman

Indexer

Monica Ajmera Mehta

Production Coordinators

Nitesh Thakur

Manu Joseph

Cover Work

Nitesh Thakur

About the Author

Paul F. Johnson has been writing about software since the days of the old 8-bit micros in the 1980s, with his first piece on software being published in 1984 for BBC Micro. From there, his passion to learn and develop increased, along with his love for chemistry. For many years, he married the two at the University of Salford, culminating in commercial work for RiscStation Ltd. as well as working on the award winning Scribus desktop publishing application.

For many years, he contributed a great amount of time to the Fedora Project (a community-based Linux distribution sponsored by RedHat) and, in 2002, he started becoming interested in the Mono project from Ximian. That was the beginning of the end of his time in the field of education and, over the next 10 years, he learned how to code in VB.NET. and C#. Ximian was sold to Novell, which was in turn bought out, and from that, Xamarin was born. The rest is history.

With the advent of a workable .NET system that can be used in a non-Windows environment, the stage was set for the hard work to begin. When Xamarin released monodroid, he could see that his years of learning could now be turned into a profit. He started developing code full-time for the platform. Shortly afterwards, he began working on iOS devices as well.

At the start of 2013 he, along with Andrei and his good friend Scott, formed Sporkish, the objective being to produce unbeatable software for the mobile world.

This is his first foray into the world of books, though he has had many articles published in the Overload and C Vu journals of Association of C and C++ Users (he edited the latter for over a year). He is currently developing mobile applications for Farmtrack Pty (Australia) and HelloU (London, UK), and is in discussions with Packt Publishing on the publication of another book, this time on AI and Expert Systems.

Paul is 42, lives with his wife, dog, cats, and son, and drinks way too much coffee!

Acknowledgments

There are many people I have to thank for their assistance with this book. My biggest thanks has to go to my wife Becki and son Richard, who have had to put up with me working three jobs at once in order to get this done. I should also mention my daughter Ashleigh, for bringing me biscuits and fizzy drinks while I worked.

I'd like to thank my parents for the great lesson that the best way to get ahead in life is to get off your backside and do it. Without them getting me my first computers, this book would never have been written and I'd still be in some awful FE College somewhere, teaching the same things day after day.

Jock and William at Farmtrack have been a great source of inspiration, without whom this book would probably have not been written by me – they really gave me my iOS break; so thanks guys, this is down to you and your belief in me.

I must also thank Scott. He probably doesn't know how much of an influence he has been on me in these last five years. His hand has guided, shaped, and smacked me when I needed to be and without his encouragement I would have never have left education. All this, when I have never even met him. I will one day, and until then, our 5-hour Skype calls shall continue.

Have I missed anyone? Well, in no particular order...

Willow, John King (Southport College), Dr. Derek Bloor (University of Salford; sadly, no longer with us), Dr. Trevor Crowley, Roger Darlington, Dr. Andrew Hill, Trevor Green (HelloU), Neil Hewitt (South Cheshire College), Yogesh and Akash at Packt Publishing for being patient, and Steve Hopley and Chris Ignatius (St Helens College). Roy Heslop at CTA Direct deserves a big mention for helping me out when my computer decided to die. And not forgetting Andrei either!

This book is dedicated to anyone who wants to learn and wants to progress.

Enjoy!

Paul

Newton-le-Willows, 10th Sept 2013

About the Reviewers

Yaroslav Bigus is an expert in building cross-platform web and mobile apps. He has over four years experience in development and has worked for companies in Leeds and New York. He has been using the .NET Framework stack for developing back-end systems, JavaScript for front-end systems, and Xamarin for mobile devices.

He has also worked at CTN Systems (New York, USA) and Fluid Four (Leeds, GB). He is now working for an Israeli startup called yRuler.

I'd like to thank my mother Lesya for all the love and attention that she showered on me to make me a good human being. Also, my dear friends and my lovely woman for being with me during hard times.

John Goodwin was born in 1979 on South Korean soil as an American citizen to US Army parents. He moved a lot with his family, eventually spending much of his youth in Washington State.

After meeting his wife Jane, he moved to California, where he soon became employed as a professional software developer for a company in Canoga Park, CA, known as Cyberspace Headquarters, LLC. While working there for several years, he progressed from the the new guy to Lead Software Developer, in charge of two or sometimes three other software developers as well as off-shore development projects. The economic downturn of IT companies post 9/11 eventually took its toll, and he moved with his wife further out of the city to look elsewhere for employment.

Next, he took up some teaching opportunities in the rural northern Los Angeles County, and was also a short-term employee at a Simi Valley factory, looking to improve worker efficiencies. Soon, he heard of an opening in the city of Los Angeles as a contract software developer; he was interviewed and started work.

During the housing boom from 2002 to 2006, it was very clear that Southern California's bubble was about to burst. He and his wife sold their home in favor of moving to Lake Royale, where he continued to work for the City of Los Angeles by telecommuting.

After working for seven years for the City of Los Angeles, he started working at CareAnyware, developing healthcare related software for home health and hospice.

CareAnyware was soon after purchased by Brightree, where he continues to work, writing post-acute healthcare software with a great team.

His passion for bringing technology to bear on creating value in the lives of others is mysteriously tolerated by his loving wife.

Unable to find a normal way to work out and keep fit, he participates in local sprint triathlons (and maybe he will participate in a half triathlon in 2014), which motivates his workout schedule.

His brother, mother, and stepfather all contribute to his family support structure.

Andri Yadi is a developer, entrepreneur, influencer, and educator in the IT industry, especially the mobile apps field. As a developer, he has been developing in many well-known programming languages since he was 16. Since iOS SDK was first released in 2008, he's been one of the early adopters of Objective-C and iOS SDK. As an entrepreneur, he has founded four software companies since 2003. The last one is PT. Dycode Cominfotech Development (DyCode), where he put all his heart, time, thoughts, and passion for the last 6 years. As an influencer, he has been actively influencing the mobile apps industry and the developer community in Indonesia. He co-founded four developer communities; one of them is the ID-Objective-C community, Indonesia's first and biggest iOS developer community, where he also serves as the president. For his technical expertise and community influence, he's received the Microsoft Most Valuable Professional (MVP) award 6 years in a row. As an educator, he has been delivering more than 100 speeches and training. Lately, he's been actively talking about iOS and the Microsoft Windows Azure development, and delivering regular iOS app development training.

This book is dedicated to knowledge seekers; ones who always stay hungry and stay foolish.

I'm very thankful to my future wife, Gina Rizka Ariany, for her incredible support. I also thank my parents for their unconditional love and everything else. Thanks to DyCoders, for supporting me when reviewing this book, my fellow community for sharing, and fellow professionals for challenging me.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

For Becki,

This book is for you. You have stood by me through this, kept me on track and loved me throughout. I can never thank you enough for what you have done and for that, this work is for you. Thank you my love.

Table of Contents

Preface	1
Chapter 1: Installing the Xamarin Product Range for Android and iOS	5
Installing Xamarin.iOS and Xamarin.Android	5
Downloading the software	6
Installing the software	6
Enabling Visual Studio to build and run iOS applications	6
On the Mac	7
On the PC	7
Installing additional code for Android development	10
For iOS users	10
Summary	11
Chapter 2: The User Interface	13
Creating the User Interface with Xcode	13
Screen origins and sizes	18
MonoTouch.Dialog (MT.D)	18
Changing the keyboard type	20
Using ShouldReturn	20
Using ResignFirstResponder	21
Adding a toolbar to the keyboard	21
Creating your own Pickers on MT.D	25
UITableView and UITableViewCell	29
Colors, buttons, and labels	29
Ensuring you have the correct size bounding boxes	30
UIColor	30
UIButton	31
UIControlStates	33
Summary	33

Chapter 3: Views and Layouts	35
Selection of the project type	35
Application types and their view types	36
The iOS layout	36
The Canvas model	36
How to avoid some of these problems	37
Views and View Controllers	37
Other Views	38
Activity Indicator and Progress View	39
UIImageView	40
UICollectionView	41
UIWebView	42
MapView	44
UIScrollView	44
AdBannerView	45
Implementing a view with multiple View Controllers	45
Summary	46
Chapter 4: Controllers	47
UITableView and UITableViewCell	47
Creating a read-only table	49
UITableViewCell	51
Reusable cells within a table	52
Sections and Rows	52
Indexes on a TableView	53
Navigation with UITableView	53
Within code	53
With Xcode	54
Navigation using UITableView	56
Returning to the RootView	57
TabBars	57
Handling the Tab Bar in code	58
PageControl	59
GLKit	59
Summary	59
Chapter 5: UI Controls	61
Controls and widgets	61
UI Controls	62
Control selection	62
UIButton	63
UIStepper	65
The other controls	66
Comparing Android to iOS UI controls	67
Summary	68

Chapter 6: Events	69
Handling events	69
Delegates	69
Attaching an event to multiple controls	70
Synchronous versus asynchronous event handling	71
Synchronous walk	71
Asynchronous walk	71
In a programming context	71
Events and controls reference	73
Other significant control events	73
AVAudioPlayer and AVRecordClass	74
AVAudioSession	74
ABAddressBook	75
ABNewPersonViewController	75
ABPeoplePickerNavigationController	75
ABPersonViewController	76
ABUnknownPersonViewController	76
AudioConverter	76
AudioSession	76
InputAudioQueue	77
OutputAudioQueue	77
AUGraph and AudioUnit	77
AudioConverter	77
CAAnimation	77
CBCentralManager	78
CBPeripheral	78
CBPeripheralManager	79
CFSocket	80
CFStream	80
CLLocationManager	80
MidiClient	81
MidiEndpoint and MidiPort	81
Monotouch.Dialog	81
BadgeElement, BaseBooleanImageElement, GlassButton, LoadMoreElement, MessageElement, and StringElement	82
BoolElement	82
DateTimeElement	82
DialogViewController	82
EntryElement	83
StyledStringElement	83
EKCalendarChooser	83
EKEventEditViewController and EKEventViewController	83
EAAccessory	84
The NS classes	84
NSCache	84
NSKeyedArchiver	84
NSKeyedUnarchiver	85
NSNetService	85

Table of Contents

NSNetServiceBrowser	85
NSStream	86
GLKView	86
GK classes	86
GKAchievementViewController, GKFriendRequestComposeViewController, and GKLeaderboardViewController	86
GKGameCenterViewController	86
GKMatch	87
GKMatchmakerViewController	87
GKSession	87
MKMapView	88
MPMediaPickerController	89
MFMailComposeViewController and MFMessageComposeViewController	89
PKAddPassesViewController	89
QLPreviewController	89
SK classes	89
SKProductsRequest	90
SKRequest	90
SKStoreProductViewController	90
UIClasses	90
UIAccelerometer	90
UIActionSheet and UIAlertView	91
UIButtonBarItem	91
UIImagePickerController	91
UIPageViewController	91
UIPopoverController	92
UIPrintInteractionController	92
UIScrollView	92
UISearchBar	93
UISplitViewController	93
UITabBar	93
UITabBarController	94
UITextField	94
UITextView	94
UIView	95
UIWebView	95
Ad classes	95
AdBannerView	96
AdInterstitialAd	96
OpenTK	96
IGameWindow	96
iPhoneOSGameView	97
Summary	97
Chapter 7: Gestures	99
Gestures	99
Gesture code	101
Types	102

Adding a gesture in code	102
Continuous types	102
Other UIGestureRecognizerState values	104
Handling drag-and-drop	104
Summary	105
Chapter 8: Threading	107
<hr/>	
Threading Concepts	107
The main UI thread	108
Deadlocking	108
Avoiding deadlocks for synchronized accessors	109
Starting a new thread from the main UI thread	109
Using locks	112
The AppDelegate class	114
Summary	114
Chapter 9: Threading Tasks	115
<hr/>	
A brief introduction to threading	115
Using background threading within your app	116
BackgroundWorker	116
ThreadPool.QueueUserWorkItem	119
Using System.Threading.Tasks	120
Problems while using Tasks on threads	120
Using Asynchronous code	121
Tasks and EventHandlers	121
A more practical example	121
Summary	122
Chapter 10: Animation	123
<hr/>	
Handling bitmaps	123
Scaling the image	123
Rotating the image – Part 1	124
Underpinning bindings	125
Analysis of the code	125
Freeing memory after use	126
Rotating the image – Part 2	128
Summary	128
Chapter 11: Handling Data	129
<hr/>	
Using SQLite	129
Installing and setting up SQLite	129
Database basics	130
A simple database class	130
Create a connection to the database	131

Setting up an SQLite helper class	132
Writing helper class methods	133
Adding data to the database	134
Data manipulation using LINQ	135
LINQ – a whistle-stop tour	136
SELECT and WHERE in LINQ – a common cause of confusion	137
Using Select in LINQ	137
Replacing SQL with LINQ	138
Summary	138
Chapter 12: Peripherals	139
Using the camera	139
Accessing the camera (Xamarin.Mobile)	140
Accessing the camera (Native)	140
Saving to the Photo album (Native)	141
GPS and Mapping	141
GPS with Xamarin.Mobile	141
Calculating your speed	143
Using Core Location	143
Setting up Core Location and delegate	143
Finding where the user is	145
Adding a map	147
Adding a pin	150
Storage on the phone	150
Making a phone call	150
Sending and receiving a text message	151
Accessing the Internet	152
Multimedia	155
Playing a video	155
External URL	155
Internal source	155
From the photo library	156
Recording a video	156
To record a video	156
Saving the video	157
The audio system	157
Playback	157
Recording Audio	158
Setting up the audio NSDictionary	159
Summary	160
Chapter 13: User Preferences	161
The built-in system	162
Reading and writing to the .plist file	163

Rolling your own settings system	164
Serializing and deserializing data	164
Setting up the Settings file	165
The handler class	165
The data class	167
Summary	168
Chapter 14: Testing and Publishing	169
<hr/>	
Provisioning and signing your app	169
TestFlight	170
Provisioning	170
Registering the app	171
Creating the developer profile	172
Creating your certificate	173
Back to registering your app	174
Enabling TestFlight within Xamarin Studio	174
Registering on TestFlight	175
Inviting and registering devices	176
Building to TestFlight	178
Releasing your app	179
App checklist	180
Icon sizes	180
Preparing to package	181
Packaging your app	182
Creating the build configuration	182
The App Store Submission Process	184
Creating an archive	184
Submission via Xcode	185
The submission wizard	185
Summary	185
Index	187

Preface

Welcome to this book! What you will find between the covers of this book will hopefully set you on your way to producing your own iOS applications; not only that, it will also help you to start writing code that can be moved with the minimum fuss to Android and Windows 8 mobile phones and tablets. Hold on to your hats folks!

What this book covers

Chapter 1, Installing the Xamarin Product Range for Android and iOS, explains how to set up your PC or Mac to develop apps for your iOS device.

Chapter 2, The User Interface, deals with creating a user interface and its key components.

Chapter 3, Views and Layouts, explains how creating a user interface isn't just about putting buttons on a screen; you also need to start with the right application type.

Chapter 4, Controllers, gives the basics of the two most used forms of Navigation and View. iOS uses an MVC system (Model, View, Controller). We've had the first two, now let's see about controllers.

Chapter 5, UI Controls, explains how the controls are more than just buttons and textboxes; we can really go to town on how these bad boys look.

Chapter 6, Events, explains how, without events, your iOS device is nothing more than a lump of plastic. iOS is rich in events, and they're all here.

Chapter 7, Gestures, covers the operations that iOS makes a big play on, such as being able to pinch, sweep, and move around the screen. These operations are called gestures and they're very simple to use!

Chapter 8, Threading, deals with iOS as a multithreaded system. How these threads interact determines how an app will behave.

Chapter 9, Threading Tasks, deals with task scheduling as well as asynchronous tasks. It explains how Android works with these tasks.

Chapter 10, Animation, gives an insight into animation, as it is an important part of any app.

Chapter 11, Handling Data, explains how it is surprisingly easy to handle and manipulate large amounts of complex data within C# and iOS.

Chapter 12, Peripherals, explains how to code to take advantage of the calling and texting feature on your phone as well as use the GPS system.

Chapter 13, User Preferences, explains how to store your settings for the built-in system as well as create your own cross-platform settings code. Storing your settings is a very important part of any app.

Chapter 14, Testing and Publishing, deals with how to test and publish the app after you have finished developing it.

What you need for this book

A Mac (running OS.X Lion or Mountain Lion) with Xcode installed and a copy of Xamarin.iOS (you can use the free download version from www.xamarin.com). If you're using a PC, you'll need to be running Windows 7. It will need a Mac somewhere on the network to deploy and use Xcode from.

An iPhone or iPad is also useful.

Who this book is for

This book is for those who already code in C#; it is the basic assumption used. You need not have ever written anything for a mobile device or have coded in Objective C. If you're interested in developing code for iOS, then you're in the right place.

Conventions


In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.


Code words in text are shown as follows: "A `UIImageView` has no events attached to it, so if you need an image that can be clicked on, they can be planted onto a `UIButton` class "

A block of code is set as follows:

```
var r = new UIButton();
r.Frame = new RectangleF(0, 0, 100f, 100f);
// button 100 x 100 at 0, 0
var i = new UIImageView(new RectangleF(15f, 2f, 70f, 70f));
i.Image = UIImage.FromFile("path/toimage.png")
    Scale(new SizeF(70f, 70f));
var l = new UILabel(new RectangleF(2f, 78f, 96f, 20f));
l.Text = "Hello world";
r.AddSubview(i);
r.AddSubview(l);
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The class needs to be defined in the connector as both an outlet and action (with the **Event** selected to be **Value Changed**)."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Installing the Xamarin Product Range for Android and iOS

The Xamarin product range covers iOS, OS X, and Android development. This enables the .NET Framework development on devices that do not natively support it via the respected and mature Mono framework.

In this chapter we will cover the following topics:

- Installing Xamarin.iOS, Xamarin.Android, and Xamarin Studio
- Setting up a Windows machine to develop apps for iOS

Installing Xamarin.iOS and Xamarin.Android

Installing Xamarin for Windows and OS X is a very similar and simple process. Before you download, you will need to ensure that your computer has the following requirements as the minimum specifications:

Windows	Mac
<ul style="list-style-type: none">• Windows 7 or Windows 8• Visual Studio 2010 or 2012	<ul style="list-style-type: none">• OS X Lion or Mountain Lion• Xcode v4.6 or above

For both, the general rule is the more memory you have the better. You also need to have a live network connection.

Downloading the software

The website is able to check the operating system you are using and, when you select download, the correct version for your operating system will be downloaded.

You will need to give Xamarin some basic information before you are allowed to download the evaluation copy. The evaluation will allow you to develop and deploy apps for 30 days, after which you will need to purchase a copy. If you have placed any apps on the Apple store or Google Play, they will no longer function.

On a PC, double-click on the `XamarinInstaller.exe` file (Windows 8 users should run this file as an administrator). For Mac users, double-click on the installer.

Installing the software

Depending on your network connection, this can take anywhere up to an hour as each package is downloaded in turns and then installed. The installation process is automatic, and anything required for the software to work is installed with the exception of **Xcode** – it is simple to check whether you have this installed on your Mac: click on the `Applications` directory and look near the bottom for the icon. If it's not there, go to the App Store and type in `xcode`. The download is free. Unless you have a reason to change the defaults, accept the default setup options by clicking on **Next** each time an option is presented.

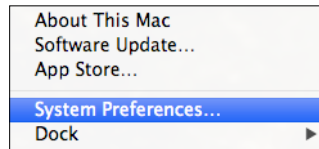


Enabling Visual Studio to build and run iOS applications

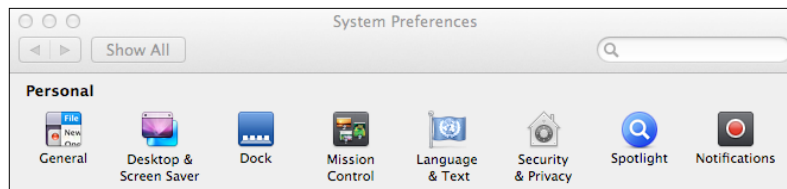
For Windows to create, build, and deploy iOS applications, it must be connected to a Mac somewhere on a network. The Mac must also have Xamarin.iOS installed. This is a two-part process.

On the Mac

1. Click on the Apple icon and select **System Preferences**.



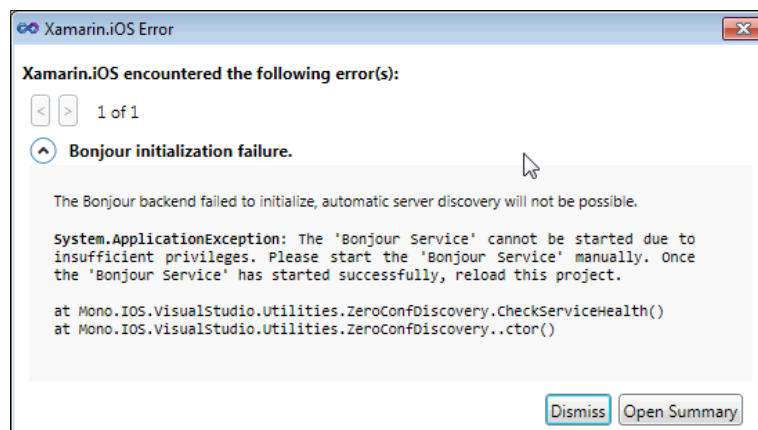
2. Then you need to select the **Security & Privacy** preference option.



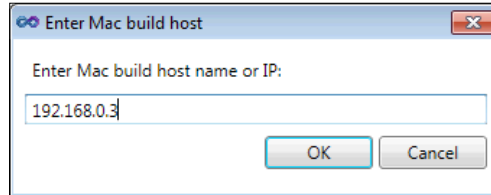
3. Then the firewall will need to be switched off. While this is not normally a good idea, a majority of internal networks have a sufficiently good firewall at any router. If you are not happy with this, leave the setting—you just won't be able to run or develop for iOS under Windows.

On the PC

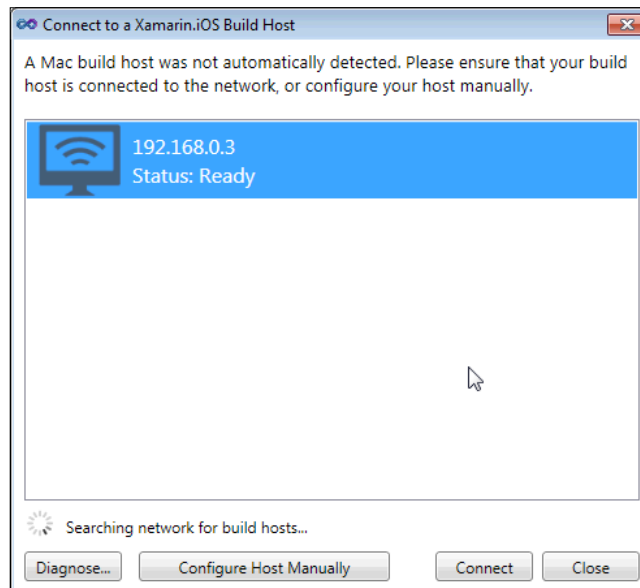
When Xamarin.iOS for Visual Studio is installed, it also installs a small listener service called **Bonjour**. When you try to create an iOS application, Bonjour will attempt to automatically find a Mac on your network. This may fail, and if it does you will see the following screenshot:



You don't need to worry about the error. Click on **Dismiss**. You will be presented with a window that allows you to enter the IP address of your Mac.



In this example, the IP address from the Mac on my network has the address shown in the preceding screenshot. When you click on **OK**, Bonjour will attempt to attach to Mac. If it is successful, you will be presented with the following screenshot:



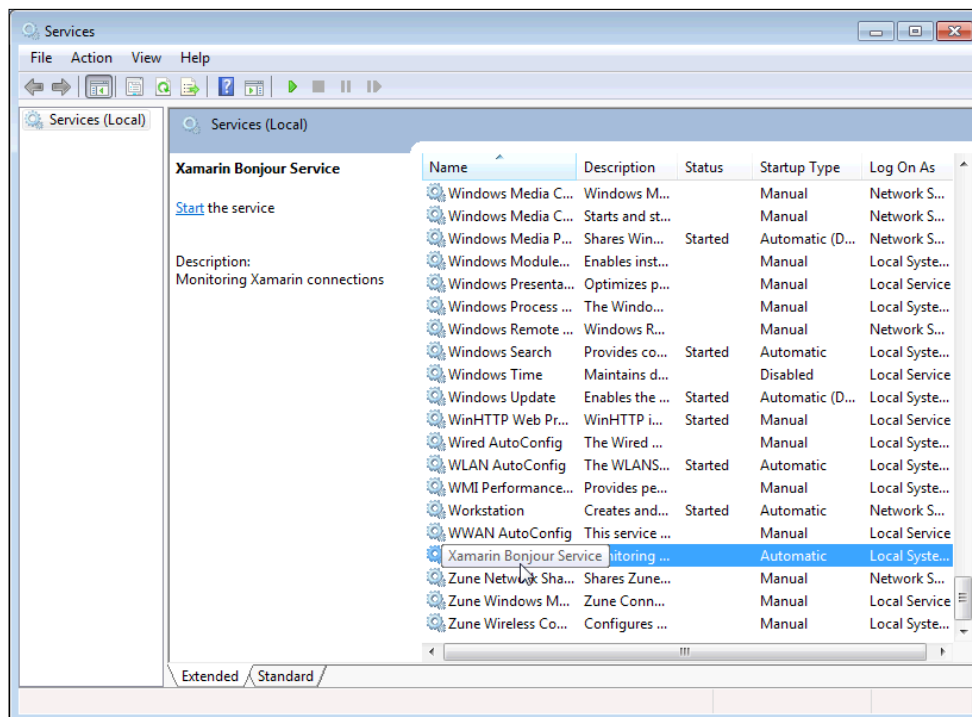
As soon as you see the window shown in the preceding screenshot on your screen, you are nearly there. The next step is to click on the host machine (highlighted in the preceding screenshot) and click on **Connect**. Once done, you're set up on the PC to develop for iOS under Windows.

All of the tools for development (like the Simulator – a simulated iOS device) can be either on one of the iPhone or iPad ranges).

If the Bonjour service was unable to automatically determine the Mac, you may need to manually set up the service. This can be performed very easily.

Assuming you're on Windows 7, the following steps will help you set up the Bonjour service:

1. Click on the **Start** button and select **Control Panel**.
2. On the **Control Panel**, click on **Administrative Tools** and from there select **Services**. You will be presented with a window, as shown in the following screenshot:



3. Ensure that the status is **Started** and the startup type is **Automatic**. (This will start the Bonjour service when Windows is restarted.)

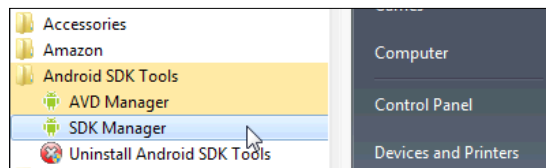


When using a PC to develop iOS applications, ensure that both the PC and Mac are running the same version of Xamarin.iOS—if they are out of step, you will need to install a newer version of Xamarin.iOS on the device that is out of step.

Installing additional code for Android development

This is different under iOS and PC but the end effect is the same.

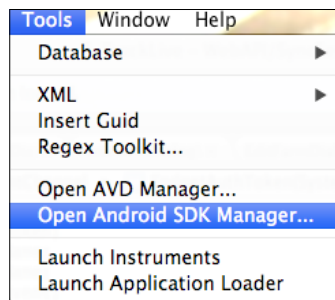
When Xamarin.Android is installed on either platform, the compiler and the minimal set of the **Android Software Developers Kit (SDK)** is installed. This will let you get going with the development but not allow you to target a range of devices. It is, therefore, important to install the SDK for other versions of the Android operating system. This is preformed using the **Android SDK Manager**.



To access **SDK Manager** on a PC, select **Start** and on the **All Programs** menu there is a menu option called **Android SDK Tools** under which is the **Android SDK Manager**. Select the **SDK Manager** and you will be presented with a new window that allows you to select the SDK you want.

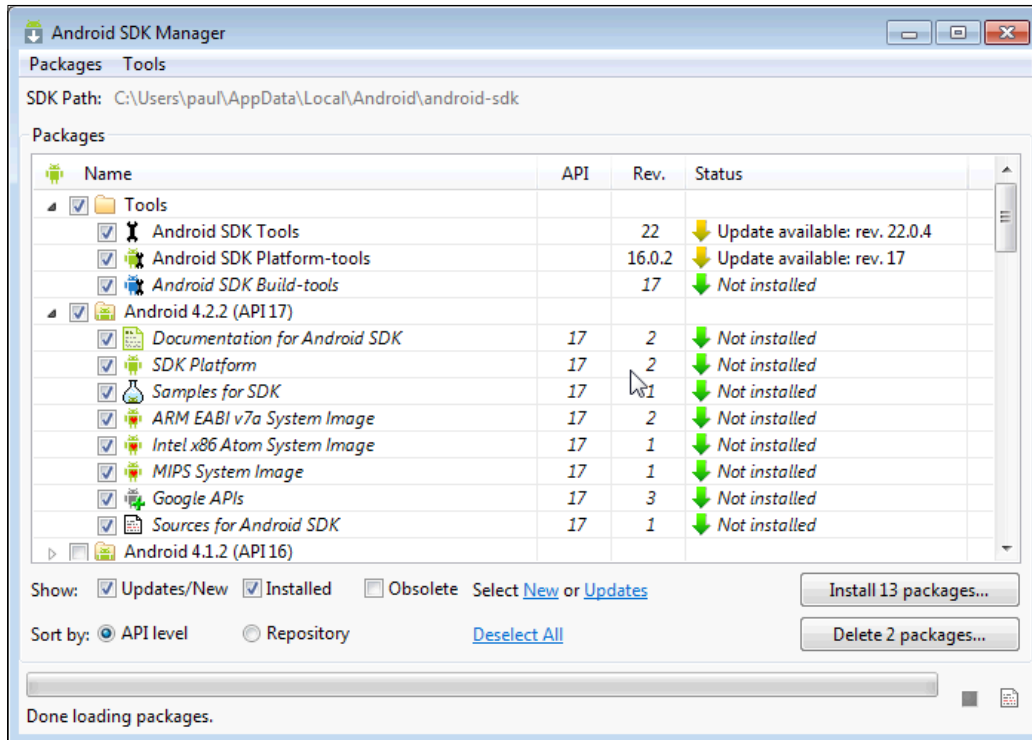
For iOS users

A part of the installation process on a Mac installs the **Xamarin IDE (Integrated Development Environment)** – Xamarin Studio. This is similar in many ways to Visual Studio and performs a very similar job.



To access the SDK Manager on Xamarin Studio, select **Tools** followed by **Open Android SDK Manager**.

In both cases, you will be presented with the following screenshot from the SDK Manager. It is simplest to select all the SDKs and click on the **Install packages...** button to start the process. Depending on your network connection speed, this process may take a while.



Summary

That's it—you're set up on both a Mac and PC to create amazing applications for Android and iOS. Your development environments are set up. For the rest of this book, though, I will concentrate solely on the development of iOS applications and leave Android to the companion book.

2

The User Interface

A user interface is the primary method of communication between a device and the user. The design and appearance is what differentiates a good app from an amazing one.

In this chapter, we will be covering some of the essential features of the user interface:

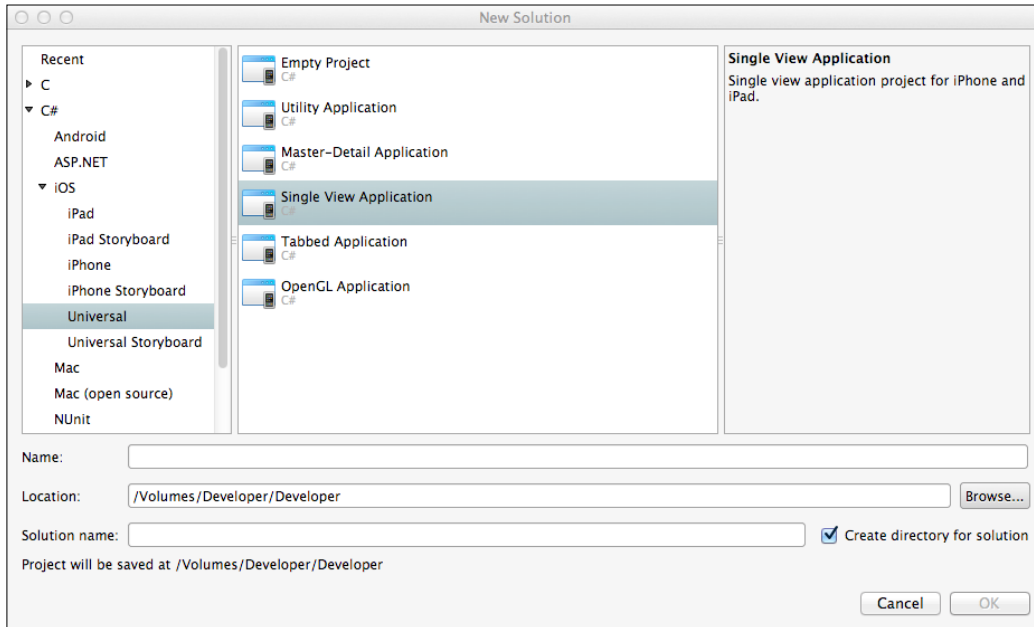
- Canvas
- `MonoTouch.Dialog`
- Incorporating external views into your user interface
- Colors
- Labels
- Images

Creating the User Interface with Xcode

With the exception of `MonoTouch.Dialog`, the user interface for any iOS application is created using Xcode.

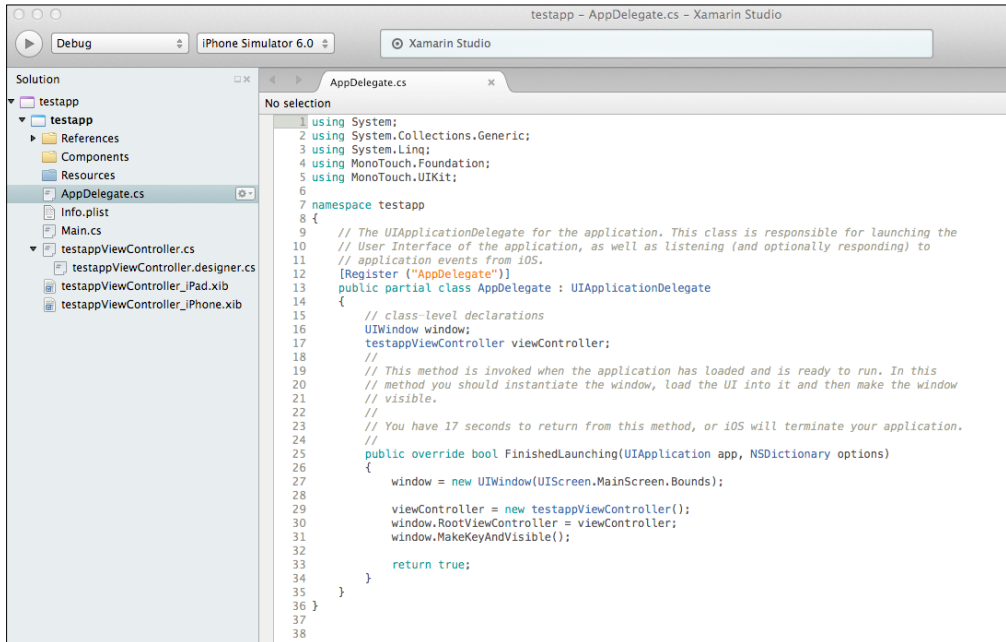
When considering how to create a user interface and the positioning of any of the available widgets, you need to think along the lines of fuzzy felt placed on a canvas. You are able to place any iOS widget anywhere on the felt.

To create a simple user interface, create a new iOS application. Click on **File** then **New**. You will be presented with a window as follows:



Click on **Single View Application**, enter a filename in the **Name** field, and click on **OK** once it is done. Xamarin.iOS will create a directory containing all of the folders and files required to get you started. Unchecking the **Create directory for solution** box will still create the application, but the files will not be held in a directory structure, but instead wherever the **Location** path points to. The application can still be edited and worked on, but there is a good chance that files will be lost or overwritten (for example, if you work on multiple projects, each project will create an `AppDelegate.cs` file. When one is created, the new version will overwrite the old one.)

Once the application structure is set up, you will be presented with a new text editing window shown as follows:



```

1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using MonoTouch.Foundation;
5 using MonoTouch.UIKit;
6
7 namespace testapp
8 {
9     // The UIApplicationDelegate for the application. This class is responsible for launching the
10    // User Interface of the application, as well as listening (and optionally responding) to
11    // application events from iOS.
12    [Register ("AppDelegate")]
13    public partial class AppDelegate : UIApplicationDelegate
14    {
15        // class-level declarations
16        UIWindow window;
17        testappViewController viewController;
18        //
19        // This method is invoked when the application has loaded and is ready to run. In this
20        // method you should instantiate the window, load the UI into it and then make the window
21        // visible.
22        //
23        // You have 17 seconds to return from this method, or iOS will terminate your application.
24        //
25        public override bool FinishedLaunching(UIApplication app, NSDictionary options)
26        {
27            window = new UIWindow(UIScreen.MainScreen.Bounds);
28
29            viewController = new testappViewController();
30            window.RootViewController = viewController;
31            window.MakeKeyAndVisible();
32
33            return true;
34        }
35    }
36 }
37
38

```

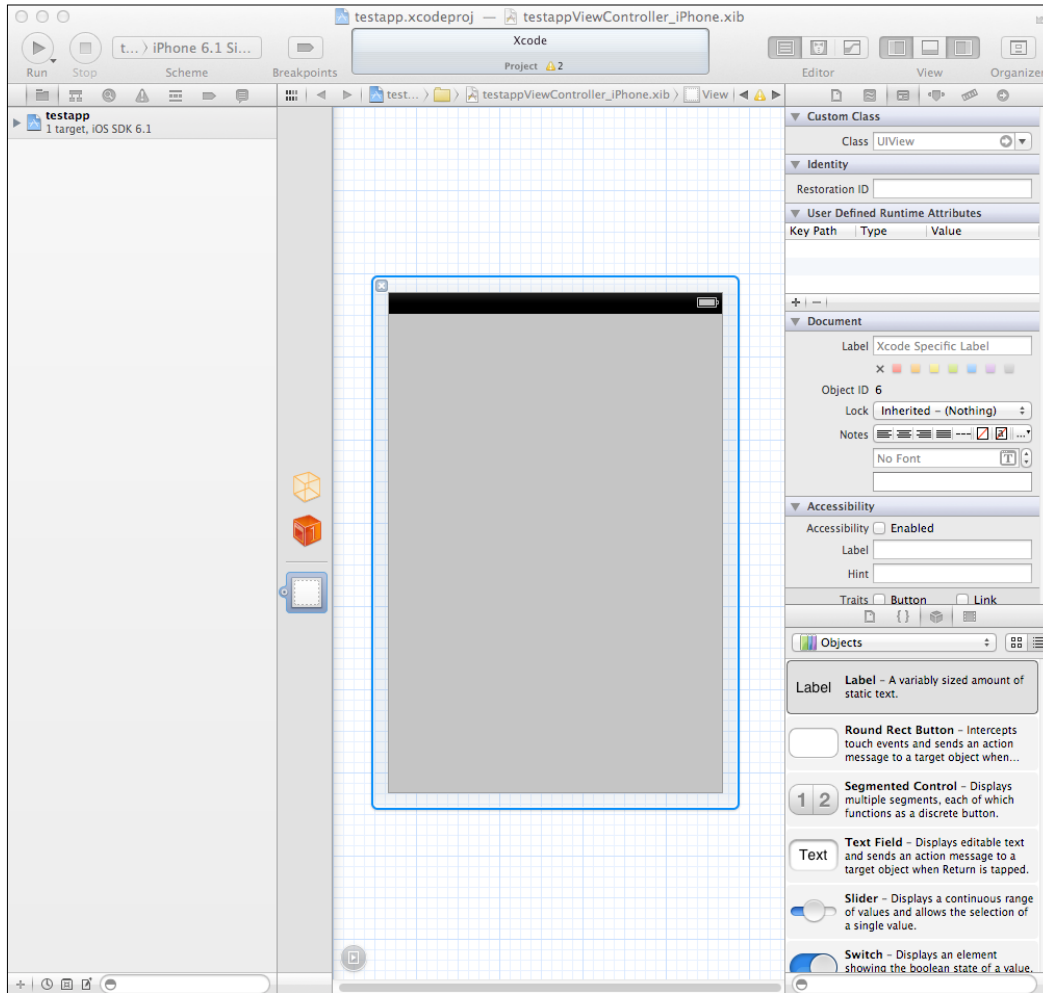
At the bottom of the solution explorer, there are three files that need attention:

- **testappViewController.cs**: This is the file for creating the C# code for your application
- **testappViewController.designer.cs**: This is a designer file created by Xamarin.iOS and is based on the user interface created with Xcode
- **testappViewController_iPad.xib** and **testappViewController_iPhone.xib**: Any file with a `.xib` extension is an Xcode designer file



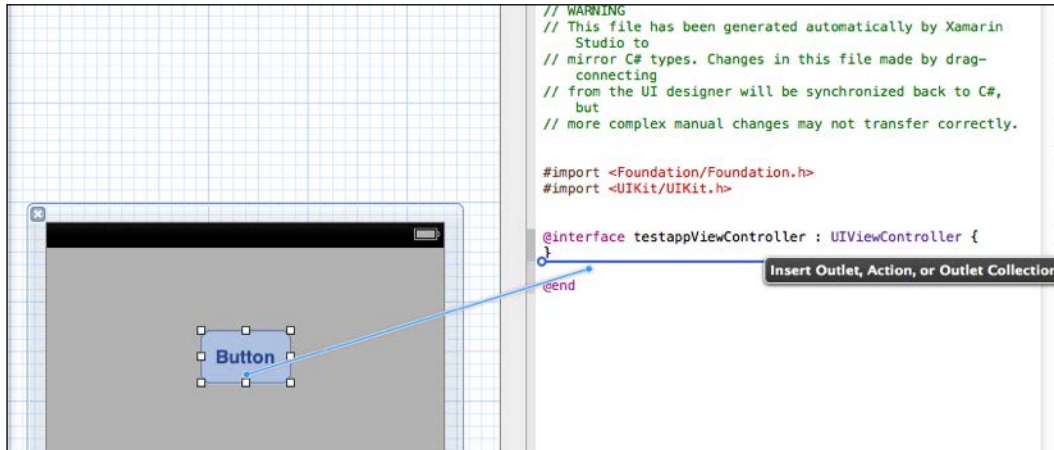
To start Xcode, simply double-click on the `.xib` file. To edit the `.xib` file, you must double-click on either the **testappViewController_iPad.xib** or **testappViewController_iPhone.xib** file depending on the view controller user interface you wish to edit.

Xcode is quite a simple designer, but in that simplicity is a very powerful piece of software. On startup, you will see the following screen:



The Xcode designer is a very complex piece of software. However, because there are books written on how to use it to its maximum potential, and due to space constraints, I will limit the discussion here to the minimum.

To add a widget, simply select and drop it into the main view. You can drop any widget more or less anywhere. However, your application won't know anything about it at all as it needs to be connected. Connecting a widget is simple. The preceding screenshot shows a button called Connector. Click on this icon and another frame will appear to the left of the properties frame. To connect a widget, click on *Ctrl* and drag it to the Connector frame.



From this point, you have to decide the type of the button: **Outlet**, **Action**, or **Outlet Collection**. Each is very different and their names are sometimes confusing.

An **Outlet** button would be normally considered for displaying information and not accepting events. However, actions, say, click events, are handled in **Outlet**. If you connect a button as **Outlet**, it is considered an interface. It is open to all modifiers and events available for that object. **Outlet Collection** is a collection of outlets. **Action** is just that—it is a specific action (or event) linked to that object.

There is a major difference, though, between them. If you have a button connected as **Outlet**, events have to be specifically added. If you don't do anything with the button at all, the application will run on your device, but the button will do nothing. If you have the button connected as **Action**, the code for the action must be written before the application is run. Failure to do so will result in the application crashing.

Events will be dealt with in a later chapter.

Screen origins and sizes

All screens and views start with 0,0 at top left. To obtain the screen size (remember, this is going to be different for different versions of iPhones and iPads), consider the following lines of code:

```
var window = new UIWindow(UIScreen.MainScreen.Bounds);
float ScreenX = window.Screen.CurrentMode.Size.Width;
float ScreenY = window.Screen.CurrentMode.Size.Height;
```

MonoTouch.Dialog (MT.D)

iOS screens usually contain a lot of data in a list form (think of how Facebook or Twitter looks). On iOS, these are constructed using `UITableView`. This is a very flexible piece of the UI, but can be tricky to code for. To alleviate the problems with `UITableView`, Xamarin created the `MonoTouch.Dialog` class. The benefit of `MT.D` is that Xcode is not required for designing the interface so it can just as simply be created under Windows as well as Mac.

`MonoTouch.Dialog` views are very simple to create and work on a three-tier system for design:

- **Elements:** These contain the likes of `on/off` Boolean switches, strings, images, and anything else you would normally see in a user interface.
- **Sections:** These hold any number of elements.
- **Roots:** These hold the sections. An `MT.D` class must have at least one root element.

There are many different types of elements. In the following example, a simple user interface is constructed (this code is autogenerated from Xamarin Studio when you ask it to create an `MT.D` class):

```
public partial class Login : DialogViewController{
    public Login() : base (UITableViewStyle.Grouped, null){
        Root = new RootElement("Login"){
            new Section ("First Section"){
                new StringElement ("Hello", () => {
                    new UIAlertView ("Hola", "Thanks for tapping!", null,
                        "Continue").Show();
                }),
                new EntryElement("Name", "Enter your name", string.Empty)
            },
            new Section ("Second Section"){
            },
        },
    }
}
```

```
    };  
  }  
}
```

When compiled, the code produces the following (the first view is the initial display; the second view is displayed when the **Hello** entry is clicked on):



Image A

We would get the following image after clicking:

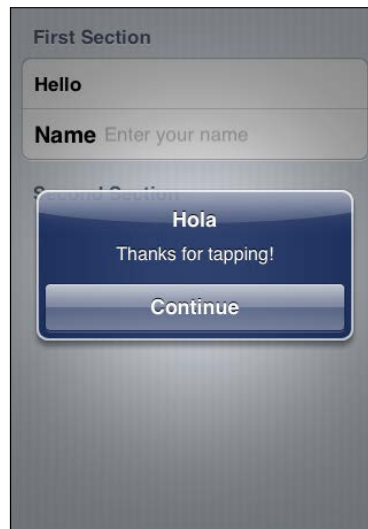


Image B

The keyboard does not have to be a standard QWERTY keyboard. It can be one specific for e-mail addresses, phone numbers, or just numbers. This is defined using `UIKeyboardType`.

Changing the keyboard type

Consider the following line of code:

```
var entryExample = new EntryElement("Caption",
    "Type a number here", string.Empty);
```

When the preceding line of code is attached to an `MT.D` class view, tapping the element will bring up a standard alphanumeric keyboard. For standard entry, this is fine; in this example, though, a different type of keyboard is required. To set this, either of the following can be used:

```
entryElement.KeyboardType = UIKeyboardType.DecimalPad
entryElement.KeyboardType = UIKeyboardType.NumberPad
```

There is an issue, though, with these keyboards and that is dismissing them. For the likes of a standard alphanumeric keyboard, a return key can be added to the keyboard itself using:

```
entryElement.ReturnKeyType = UIReturnKeyType.Done
```

The likes of the numeric keyboards do not have a return key, even if `ReturnKeyType` is added. There are three ways to sort this:

- `ShouldReturn`
- `ResignFirstResponder`
- Add a toolbar to the keyboard with a done button that dismisses the keyboard

Using `ShouldReturn`

This is a simple method to use but relies on a return key being on the keyboard:

```
entryElement.ShouldReturn += delegate {
    adminPhone.ResignFirstResponder(true); // animated
};
```

Using ResignFirstResponder

Unless it is defined in the designer or in code, the `FirstResponder` method is whatever control is clicked on first. When something else is clicked (say another `EntryElement` but this also applies to any other control), that needs to become the `FirstResponder` control. By issuing a `ResignFirstResponder` method, the keyboard for that control is closed. Clicking onto a new control should issue a `BecomeFirstResponder` control and its keyboard appears (assuming that a keyboard is associated with the control).

Adding a toolbar to the keyboard

Toolbars are not actually part of the keyboard but can be attached to the keyboard to provide additional or missing functionality to the keyboard. Adding the keyboard is different for an `MT.D` class than for a standard `UITextField` or `UITextView` control.

For an `MT.D`

Here, a subclass of the `EntryElement` class will be needed to implement the `InputAccessoryView` method. While the cell being used can be found by looking at the `TableView` from the `Root` constructor, the `InputAccessoryView` from this is a read-only parameter so it cannot be set.

An example of the subclass would be as follows:


```
public class ToolbarKeyboardEntryElement : EntryElement{
    private UITextField textField;
    public ToolbarKeyboardEntryElement(string caption,
        string placeholder, string value) :
        base(caption, placeholder, value) {
    }

    protected override UITextField CreateTextField(
        System.Drawing.RectangleF frame) {
        textField = base.CreateTextField(frame);
        UIToolbar toolHigh = new UIToolbar() {
            BarStyle = UIBarStyle.Black, Translucent = true
        };
        toolHigh.SizeToFit();
        UIBarButtonItem doneHigh = new UIBarButtonItem("Done",
            UIBarButtonItemStyle.Done, (ss, ea) => {
```

```
        textField.ResignFirstResponder();
    }
    );
    toolHigh.SetItems(new UIBarButtonItem[] { doneHigh }, true);
    textField.InputAccessoryView = toolHigh;
    return textField;
}
private NSString key = new NSString("CustomEntryElement");
protected override NSString CellKey {
    get {
        return key;
    }
}
}
}
```

The `EntryElement` class in the main code would need to be altered to read.

```
var entry = new ToolbarKeyboardEntryElement("Caption",
    "Enter a number", string.Empty);
```

 For this example, I have not included the password parameter.

For a standard `UITextField`

Here, the toolbar (as described in the preceding section) is created but without the action sheet, and then added to the `UITextField` method using:

```
var txtField = new UITextField();
txtField.InputAccessoryView = toolBar;
```

Image B (section *MonoTouch.Dialog (MT.D)*) shows the `UIAlertView` control. This is a customizable alert box that is usually used for information (for example, errors, during a slow process to stop the user from getting worried, or if a user choice is required).

The real beauty of *MT.D* is that it removes the tedium associated with the `UITableView` control. It gives the developer the majority of the facilities required from the `UITableView` control without having to mess about. Also, if you need something special (such as a standard interface button), these can be achieved simply by creating a subclass of an element type.

The basic element types supported in an `MT.D` class are as follows:

Element	Uses	Caveats to use
<code>ActivityElement</code>	Used to show that something is happening (it's a spinner)	
<code>BadgeElement</code>	Image with text next to it	
<code>BaseBooleanImageElement</code>		Base type for the Booleans, cannot be used directly (abstract class)
<code>BooleanElement</code>	Simple on/off switch	
<code>BooleanImageElement</code>	Simple on/off switch that allows for two different images to be displayed	
<code>BoolElement</code>		Cannot be used directly (abstract class)
<code>CheckboxElement</code>	Tick next to a string when selected	
<code>DateElement</code>	Displays a date picker	This is a two-part element. Part one looks like a standard <code>StringElement</code> element but with a > symbol next to the value on the right. When clicked, a <code>UIDatePicker</code> element is displayed. The selected value is returned in the value element.
<code>DateTimeElement</code>	Displays a date/time picker	Essentially is the same as <code>DateElement</code> , except that it includes the time. The time can be set in the 12 or 24 hour clock
<code>Element</code>		Base element

Element	Uses	Caveats to use
EntryElement	Allows for data entry	The optional fourth parameter in the constructor allows for the entry to be used for passwords (set as <code>true</code> for password). The constructor takes three strings: caption, placeholder, and value. Placeholder and value can be string or <code>Empty</code> , but caption must have a value.
FloatElement	Slider bar	Values are float
HtmlElement	Caption that leads to an HTML view	This is a crossover element in that the originator is just an <code>Element</code> , but gives a <code>UIWebView</code> element when clicked.
ImageElement	Produces an image	
ImageStringElement	Produces an image with a string next to it	
JsonElement	Allows for the loading of content from a local or remote URL	
LoadMoreElement	Allows users to add more items to the list on screen	
MessageElement	Consider this as the sort of message you find on Twitter	
MultilineElement	Allows multiple lines of text to be displayed	Cannot be styled
OwnerDrawnElement	Not used directly	Must be subclassed. The <code>Height</code> and <code>Draw</code> methods must be overridden
RadioElement	A radio element that allows for a single option to be chosen from multiple choices.	Requires a radio group to be specified in the element
RootElement	Base element for the root	

Element	Uses	Caveats to use
StringElement	A simple caption on the left with a value on the right	This element can also be used as a button by providing an anonymous delegate as the second parameter (as in the example above)
StyledMultilineElement	Essentially the same as MultilineElement except can be styled	
StyledStringElement	Allow for strings to be shown using the built-in styles (such as colors, fonts, and sizes) and custom formats.	
TimeElement	Displays a time picker	Same as DateElement, but for time.
UIViewElement	UIView that can be displayed	Design UIView using Xcode.

Creating your own Pickers on MT.D

I've decided to demonstrate this using UIPickerView with UIToolBar added to the top and incorporate it within UIActionSheet.

To start off with, we need two things: an event to latch onto and the model (containing the information that UIPickerView requires). It's then a case of wiring the two together.

1. First the event:

```
public class PickerChangedEventArgs : EventArgs {
    public string SelectedValue { get; set; }
}
```

Actually, this can return anything, not just a string. But for my purposes, I'll keep it as a string.

2. Next is the model:

```
public class PickerModel : UIPickerViewModel {
    private IList<string> myValues;
    public event EventHandler<PickerChangedEventArgs>
        PickerChanged;
```

```
public PickerModel(IList<string> values)
{
    myValues = values;
}
public override int GetComponentCount(
    UIPickerView picker) {
    return 1;
}

public override int GetRowsInComponent(
    UIPickerView picker, int component) {
    return myValues.Count;
}

public override string GetTitle(UIPickerView picker,
    int row, int component) {
    return myValues[row];
}

public override float GetRowHeight(
    UIPickerView picker, int component) {
    return 40f;
}

public override void Selected(UIPickerView picker,
    int row, int component) {
    if (PickerChanged != null) {
        PickerChanged(this, new PickerChangedEventArgs {
            SelectedValue = myValues[row] });
    }
}
}
```

Not rocket science – the only important part is that the `Selected()` method from the parent class is being overridden to send back the value from the row selected; everything else overrides the default class settings.

3. To wire this into the main `MT.D` class, `EntryElement` is used:

```
EntryElement myElement = null;
```

4. A `UIActionSheet` element is also needed:

```
UIActionSheet action = new UIActionSheet();
```

5. We then create UIPickerView:

```
List<string> data = new List<string>() {"Hello",
    "This is a", "test"};
Ilist<string> iData = data;
var myPickerViewModel = new PickerModel(iData);
var MyPickerView = new UIPickerView() {
    Model = myPickerViewModel,
    ShowSelectionIndicator = true,
    Hidden = false,
    AutosizeSubviews = true,
};
myPickerView.Frame = new RectangleF(0, 100, 320, 162);
// 320 = screen x size for an iPhone 4
```

6. Next, create UIToolbar and UIBarButtonItem

```
var toolBar = new UIToolbar() {
    BarStyle = UIBarStyle.Black,
    Translucent = true,
};
toolBar.SizeToFit();
var doneButton = new UIBarButtonItem("Done",
    UIBarButtonItemStyle.Done, (s, e) => {
    action.DismissWithClickedButtonIndex(0, true);
    myElement.ResignFirstResponder(true);
});
toolBar.SetItems(new UIBarButtonItem[] { doneButton },
    true);

myPickerViewModel.PickerChanged += (object sender,
    PickerChangedEventArgs e) => {
    myElement.Value = e.SelectedValue;
};
```

7. Create the EntryElement object:

```
myElement = new EntryElement("Hello", string.Empty,
    string.Empty);
```

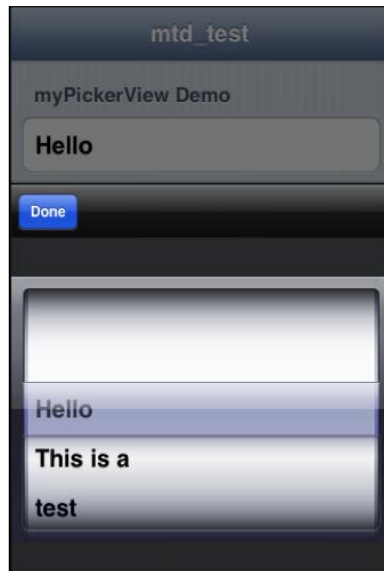
8. Now it is just a case of using the EntryStarted event to call the picker:

```
myElement.EntryStarted += (object ss, EventArgs ee) => {
    action.Style = UIAlertControllerStyle.BlackTranslucent;
    action.ShowInView(View);
    action.AddSubview(toolBar);
};
```



```
action.AddSubview(myPickerView);
action.Frame = new RectangleF(0, 100, 320, 500);
myPickerView.Frame = new RectangleF(action.Frame.X,
    action.Frame.Y - 25, action.Frame.Width, 216);
};
```

When this is all coded in, the result is as follows:



9. Subclassing an element is equally simple. The subclass `EntryElement` allows only a specific number of characters to be entered:

```
public class MaxNumberEntryElement : EntryElement {
    private UITextField textField;
    public int MaxLength { get; set; }

    public MaxNumberEntryElement(string caption,
        string placeholder, string value, int maxLength) :
        base(caption, placeholder, value) {
        MaxLength = maxLength;
    }

    protected override UITextField CreateTextField(
        System.Drawing.RectangleF frame) {
        textField = base.CreateTextField(frame);
    }
}
```

```

        textField.ShouldChangeCharacters = (UITextField t,
            NSRange range, string replacementText) => {
            int newLength = t.Text.Length + replacementText.Length
                - range.Length;
            return (newLength <= MaxLength);
        };
        return textField;
    }

    private NSString key = new NSString("CustomEntryElement");

    protected override NSString CellKey {
        get { return key; }
    }
}

```

You are still free to use `UITableView` and `UITableViewCell` of course.

UITableView and UITableViewCell

The `UITableView` method is the workhorse of the iPhone. Most, if not all, list data of whatever description is displayed using the `TableView` and `TableViewCell` methods. If you are a user of Facebook, Twitter, the standard iPhone text message application, or any form of configuration on the iPhone, you will have used these two components—this gives you an idea of how much they are used.

The topic is a massive one to cover and will be dealt with further in *Chapter 4, Controllers*.

Colors, buttons, and labels

`UILabel` is the simplest method of putting text onto a screen. It has a very limited range of actions associated with its use (for example, you cannot use it as something that is clickable). The color and text of the label can be set, as can the formatting. For example, consider the following lines of code:

```

label.Text = "text"; // sets the text label to be "text"
label.TextColor = UIColor.Blue; // sets the text color to be blue
label.BackgroundColor = UIColor.FromRGB(255,255,200);
// sets the background to be yellow

label.TextAlignment = UITextAlignment.Center;
// centres the text in the label

```

The `UILabel` method has five constructors, of which two are of great use:

- `UILabel()`
`UILabel(new CGRect(x_pos, y_pos, width, height))`

The second constructor can be replicated using the `Frame` property as follows:

- `UILabel lbl = new UILabel();`
`lbl.Frame = new CGRect(x_pos, y_pos, width, height);`

One of the main issues with using `UILabel` is ensuring that the bounding frame is large enough. There is a way to get around this:

1. Make the label much larger than required.
2. Calculate the length of the string and, using the `Frame` property, alter the size of the label.
3. Reduce the font size of the string to ensure it fits.

Both of these approaches have their benefits and disadvantages. The first is that text will always fit but only if the font size is the system default. The second is that you will always have the correct size-bounding box, but that you will have to calculate the size, and that will take time.

Ensuring you have the correct size bounding boxes

This assumes that a label has already been created using Xcode. The label in this case has a width of 96 (enough to write “More text to”):

```
string test2 = "More text to fit and boy, does it fit!";  
lblTestLabel.AdjustsFontSizeToFitWidth = true;  
lblTestLabel.Text = test2;
```

UIColor

iOS comes with a number of preset colors (such as red, green, blue, black, and white). It is also possible to create your own using `UIColor.FromRGB[A]` as well as `UIColor.FromHSB[A]` (where HSB is hue, saturation, and brightness and [A] is the alpha channel). Think of the latter as being the colors' opacity. The color can also be set from a pattern (this is useful, as it creates a color based on an image that can then be used as a brush to paint the image), `UIColor.FromImage` and `UIColor.FromCoreGraphicsColor(CI and CGColor)`, and `UIColor.FromWhiteAlpha` (a gray-scale color based on the current color space).

Using `UIColor` and `CGColor` requires a lot more legwork but does allow for greater flexibility in the colors.

The simplest to use, though, is `FromRGB`. This allows the values to be entered as byte, int, and float. The caveat here, though, is that the float values go from 0 to 1 rather than 0 to 255, so the value for 82, 184, 33 would be 0.32, 0.72, 0.13 (that is, 82/255, 184/255, and 33/255).

UIButton

A button is not just a button; it can have a whole range of interesting effects applied to it (such as the addition of graphics, a gradient color, text, and graphics). Let's assume a button (`btn`) has been created in Xcode and we wish to apply a gradient color to it.

```
var gradient = new CAGradientLayer();
gradient.Colors = new MonoTouch.CoreGraphics.CGColor[] {
    UIColor.FromRGB(115, 181, 216).CGColor,
    UIColor.FromRGB(35, 101, 136).CGColor
};
gradient.Locations = new NSNumber[] { .5f, 1f };
gradient.Frame = btn.Layer.Bounds;
btn.Layer.AddSublayer(gradient);
btn.Layer.MasksToBounds = true;
```

`CAGradientLayer` come from the `CoreAnimation` namespace.

Adding an image is also quite trivial, though the important point here is to remember that, when placing anything on a button, you have to treat that button as a new view with the origin set at the top left of the button. Remember also that a button can have a foreground and background image.

Typically, a background image will cover the entire button.

```
btn.SetBackgroundImage(UIColor.FromFile("Path/ToImage.png"),
    UIControlState.Normal);
```

The second parameter here (`UIControlState`) is the state the button (or control) is in. `Normal` is when it has not been selected. When the button is depressed, the state becomes `Highlighted` and when released, it is `Normal` again. This means you can have different images depending on the state of the button.

The foreground image will typically not cover the entire button, but will be of a particular size. For example, say the button is 92 x 92. To fill most of the button, a gap of 4 on each side would be good; this makes the dimension 84 x 84 (left and right gaps, as well as for both the height and width). To create this image for the button is a two-step process: create the image and add the image. This time, though, `UIImageView` is initially used and then fed into the `setImage` method:

```
UIImageView btnImage = new UIImageView (new CGRect(
    new PointF(4, 4), new CGSize(84, 84)));
btnImage.Image = UIImage.FromFile ("Path/ToImage.png").Scale (
    new CGSize(84, 84));
btn.SetImage(btnImage.Image, UIControlState.Normal);
```

An alternative is to add `UIImageView` as a subview to the button.

```
btn.AddSubview(btnImage);
```

The `UIButton` can also just have a color assigned to it.

```
btn.BackgroundColor = UIColor.Gray;
```

The button also comes with a default piece of text on it called `Title`. As with any text element, this can be set:

```
btn.SetTitle("Some text", UIControlState.Normal);
```

A more interesting effect is to have both text and graphics on a button. The simplest way to consider the placement is as follows:

Let `a` and `b` be the position of the top left and right of the image (in our previous example, that would be 4, 4). For ease, the same gap is on the right.

Let `c` be the offset from the top (`a + "image height" + "some gap"`).

The trick here, though, is to ensure there is enough of a gap at the bottom so it doesn't look messy.

Adding the image is a two- or three-step process.

1. If `Title` is set, clear it (this can be one in or out of the designer).
2. Add the image (see previous example).
3. Create and add a `UILabel`.

```
// step 1
if (!string.IsNullOrEmpty(btn.CurrentTitle))
```

```
btnTitle.SetTitle(string.Empty, UIControlState.Normal);

// step 3
UILabel myLabel = new UILabel(new RectangleF(4, 78),
    new.SizeF(84,10));
myLabel.Text = "some text";
btn.AddSubview(myLabel);
```

UIControlStates

There are (as previously mentioned) a number of `UIControlStates`: `Application`, `Disabled`, `Highlighted`, `Normal`, `Reserved`, and `Selected`. For most day-to-day considerations, `Disabled`, `Highlighted`, and `Normal` are the ones used most commonly.

If the `Enabled` property is `false`, the button is `Disabled`. The only problem is that this is the only way to tell if a button is disabled using the system defaults. It is probably a better idea to set the background color as well as the text when disabled.

A button does not have to be a rounded rectangle. There are four predefined buttons (`ContactsAdd`, `DetailDisclosure` [the > arrow], `InfoDark`, and `InfoLight` [the info icon with either a dark or light background]). There is also a custom `UIButtonType` type. This by default gives no border to the button, but allows for interesting buttons where a `.png` file could be the button shape. So if you want an octagonal button, you would have a `.png` file of an octagon and then write the following code:

```
btn.ButtonType = UIButtonType.Custom;
btn.SetBackgroundImage(UIImage.FromFile("octagon.png",
    UIControlState.Normal));
btn.SetBackgroundImage(UIImage.FromFile(
    "octagon-selected.png", UIControlState.Highlight));
```

Summary

As you can see from this whistle-stop tour, iOS gives you a massively rich and varied number of different objects usable within the UI. In later chapters, we will see how these can be extended and how to get the most from them.

3

Views and Layouts

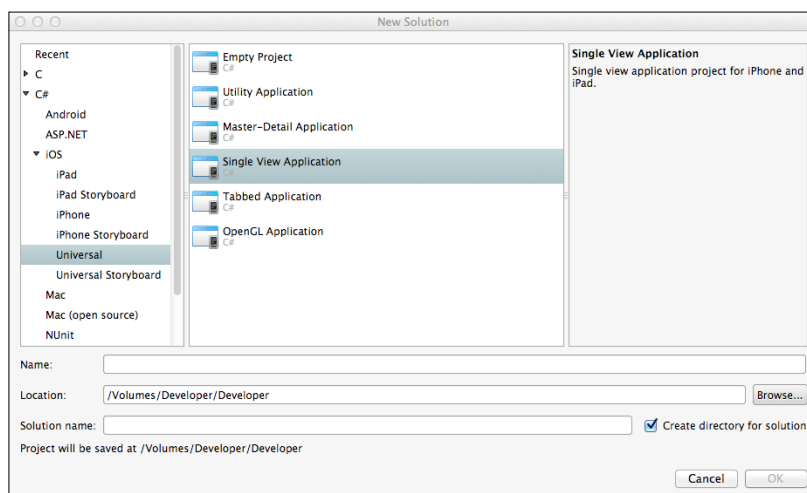
In terms of the iPhone and the iPad, a view can be thought of as what you see, but the types of views you see depend (to an extent) on the application type you select when you create your application.

In this chapter we will be covering the following topics:

- Projects and their types of layouts
- Ensuring that your design fits all iOS devices
- UI controls

Selection of the project type

When you first decide to create an iOS application, you will be presented with the view shown in the following screenshot:



Application types and their view types

The project types require a bit of an explanation, which has been provided in the following table:

Project type	View type
Master-Detail	A table based layout.
Single View	The simplest form of view. The name does not mean that you have an app with just one page, but that views don't propagate through (so view 1 may have tabs that go to view 2. View 2 could be a MT . D).
Tabbed	A standard view with a number of persistent tabs at the bottom of the screen. The view changes, but the tabs stay.
OpenGL	Fast, responsive, used for high resolution gaming. I won't be covering this here, as it is outside the scope of this book.

A more complete explanation of the project types can be found at <http://oleb.net/blog/2013/05/xcode-project-templates-difference/>.

The iOS layout

When dealing with the layout of the iPhone (and the iPad) user interface, a number of factors have to be taken into consideration; the most important is probably the physical size of the screen. While it is easy to create a user interface using Xcode, unlike Android devices, iOS does not really auto-resize. The UI is partially due to the way layouts are created in iOS.

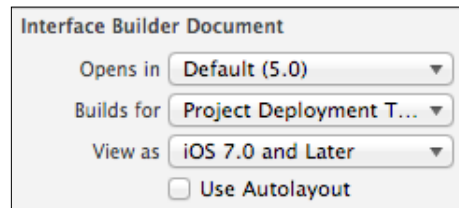
The Canvas model

When I was growing up, I was given a "fuzzy felt" kit at school. For those who don't know what it is, a fuzzy felt kit consists of a large background piece of felt onto which you stick other pieces of felt; this enables you to create lots of pictures. Designing for iOS is the same. You can drag-and-drop any piece of the user interface into the main view and leave it there. This is what gives iOS a part of the richness it enjoys; it is up to the designer to create, rather than have strict rules on what goes where. For the application, the UI elements all have an **absolute** position on the screen, rather than being relative to any other objects. Here, though, is the problem.

If you increase the screen size, those positions stay the same. So, what may look good on an iPhone 4 has bits missing on the 3G and has gaps on the iPhone 5— don't start thinking about the iPad— as the view would be just in the middle of the screen, usually squashed up.

How to avoid some of these problems

The simplest method is, when using Xcode to design your UI, you check the **Use AutoLayout** checkbox on the UI View. This does the moving around for you.



The problem here though is that you need to set this **Use AutoLayout** on every view, and it is also not supported on the iPhone 3GS. However, the 3GS is now so old that it is probably not worth going through the hoops required to auto-scale on it. The 3GS supports iOS 6, but only by hacking. iOS 5 is nearing the end of its life (at the time of writing).

Views and View Controllers

They sound similar, but they're not. The simplest way to think of a View Controller and a View is like a web page. A typical web page is a piece of information served up from a server. The content may be created dynamically (say from a database query), but for the user it's just data. This would be considered as a View. It has things on it, but no real user interaction.

A View Controller is closer to a web page constructed using `ASP.net`, or some other form of language that feeds back to a server (such as PHP). The website has a button on it. The button has an event, which is then fed back to the code (known as the handler code) behind the button. The View created using Xcode is the web page, and the source file with a connection is the server.

As outlined in the previous chapter, the objects on `UIViewController` are connected to the code behind the control by clicking on the control (as well as pressing the *Ctrl* key) and dragging the widget to the connector entry window.

Other Views

Outside of the View choice, there are a number of other views that are available, as shown in the following table:

View name	Description
Activity Indicator View	It is a modal indicator that shows that something (an activity) is occurring. This can be the loading of a web page or the rendering of a map.
Progress View	It shows the progress of time for an activity – gives the user a better idea of how long something is going to take.
Collection View	It displays a collection of cells (<code>CollectionViewCells</code>). Each cell can be defined.
Collection Reusable View	A reusable collection works as follows: say you have a group of cells which, for argument's sake, occupies the screen. In a standard collection, when the cells go off screen, they are still held in the collection. While this makes rendering faster when they return, they take up memory space. The reusable collection stores a pointer to the cell and then refreshes when back on the screen – the collection is then reused.
Table View	It will be covered in <i>Chapter 4, Controllers</i> .
ImageView	Think of this as a picture view. It has no click events and displays images. It can, however, play animations within it.
Text View	It displays multiple lines of text. Can be a read only as well as a read/write.
Web View	It is a view used to render HTML. The HTML file can be held on the phone or remotely.
Map View	It displays a map with various options. Maps are covered later in this book.
Scroll View	It is a view designed to allow more content to be accommodated on a screen than the screen size actually allows.
Picker View	It is a user-definable picker.
AdBanner View	It is an advertising banner bar for in-app advertising.
GLKit View	It is used for OpenGL-ES rendering.

I will deal with the views not covered here elsewhere in this book. As with all views, these too need to be dragged onto the view in Xcode and then linked to the main code.

Activity Indicator and Progress View

The `UIActivityIndicatorView` class is a very simple view to implement.

```
var aiActivity = new UIActivityIndicatorView()
{
    ActivityIndicatorViewStyle = UIActivityIndicatorViewStyle.Gray,
    HidesWhenStopped = true,
};
// start the indicator
aiActivity.StartAnimating();
// stop the indicator
aiActivity.StopAnimating();
```

The `UIProgressView` class is a little more complex, but still quite simple. This works on a thread system to keep track of the indicator. Let's start by setting one up:

```
var pvActivity = new UIProgressView()
{
    BackgroundColor = UIColor.Red,
    Style = UIProgressViewStyle.Bar,
};
pvActivity.SetProgress(0, true);
```

Next is to construct the thread routine. The `NSAutoreleasePool` class is used as a temporary block of memory that is released once the code within the braces has been executed. In the following code, it allows access to the `InvokeOnMainThread` method:

```
private void myTestRoutine()
{
    int n = 5;
    for (int i = 0; i < n; ++i)
    {
        Thread.Sleep(1000);
        using (var pool = new NSAutoreleasePool())
        {
            InvokeOnMainThread(delegate
            {
                pvActivity.Progress = (float)(i + 1) / n;
            });
        }
    }
}
```

And finally, link it to the Progress View:

```
Thread t = new Thread(myTestRoutine);  
t.Start();
```

UIImageView

A `UIImageView` class can bring an image in from `UIImage`, which in turn brings images in from a number of places:

- `FromFile`: a file held within the structure of the application (for example, if the application has a directory called `Graphics`, the `FromFile` would point to `Graphics | image.png`).
- `FromImage`: loads from `CoreImage` files.
- `FromResource`: loads from the `Resources` directory. These are embedded from within the application.
- `FromBundle`: loads an image relative to the main application bundle and caches it.
- `LoadFromData`: an image created from within the app.

To load a file into `UIImageView`:

```
var myImage = new UIImageView()  
{  
    ContentMode = UIViewContentMode.ScaleAspectFit;  
    Image = UIImage.FromFile("Graphics/helloXamarin.png"),  
    Frame = new CGRect(new PointF(20, 20), new.SizeF(100, 100)),  
};
```

The `UIImageView` class can also display animations. The main prerequisite for doing this is that you should have a number of images to animate. In my example, I have six images of a tractor. The wheels are the only parts that move.



To start the animation, use the following code:

```
UIImageView animation = new UIImageView()
{
    Frame = new RectangleF(new PointF(20, 20), new
        SizeF(100,100)),
    AnimationDuration = 0.5,
    AnimationRepeatCount = 0,
    Center = new PointF(animation.Center.X + 115,
        animation.Center.Y + 65),
};
animation.AnimationImages = new UIImage[]
{
    UIImage.FromFile("Graphics/track-1.png"),
    UIImage.FromFile("Graphics/track-2.png"),
    UIImage.FromFile("Graphics/track-3.png"),
    UIImage.FromFile("Graphics/track-4.png"),
    UIImage.FromFile("Graphics/track-5.png"),
    UIImage.FromFile("Graphics/track-6.png")
};
animation.StartAnimating();
```

To stop the animation, use:

```
animation.StopAnimating();
```



UICollectionView

The simplest Collection View that you're likely to see is an image gallery. Think of the Collection View as being a grid view that can be extended. Each Collection View is made up of three different items; cells, supplementary views (data-driven views), and decoration views.

Cells

Each `UICollectionView` class will contain `UICollectionViewCells`.

These cells have a main Content View (where you see something, be it a picture or the data derived within the app), and surrounding the Content View is one of the two background views: normal or selected. If the content part is not smaller than the background, the background won't be seen.

Supplementary Views

These are views that present information linked to each section of `UICollectionView`. They are data driven. Where the cells are from a data source, the Supplementary View presents that section's data (for example, the main view could be the front covers of books, the Supplementary View could be the table of contents).

Decoration View

These are not data generated and are there purely for aesthetic purposes.

Data source

The `UICollectionView` class gets its data via the `UICollectionViewDataSource` class. This class provides information, such as the cells (from `GetCell`), supplementary views (from `GetViewForSupplementaryElement`), number of sections (from `NumberOfSections` or 1 if not implemented), and the number of items per section (from `GetItemsCount`).

Cell Reuse

The `UICollectionView` class will only call the data source to get cells for items that are on the screen. Items that are not on the screen are placed in a queue to be reused.

UIWebView

The `UIWebView` class effectively transforms your device into a web browser with JavaScript capabilities as well as the usual web facilities, such as move back, forward, and typing a URL in the text field when extended with `UITextField`.

To load a web page is simple enough, as shown in the following code:

```
var web = new UIWebView();
NSURL url = new NSURL("http://www.bbc.co.uk");
web.LoadRequest(new NSURLRequest(url));
```

There are a number of factors to remember with web page loading. The first factor is that it is typically an asynchronous task; in other words, some parts are completed before others and it is quite possible that the application flow will return to the main thread before the task is completed. The second factor is the speed. I'll not concern myself with the second factor for now.

- To overcome the problem caused by asynchronous tasks, there are a number of events that can be used:
 - LoadStarted
 - LoadFinished
 - LoadError
 - The boolean IsLoading

The IsLoading boolean is a flag that can be checked at any point to determine if something is loading (true) or has completed loading (false).

For example:

```
web.LoadStarted += delegate {
    // start ActivityIndicator here, stops anything else happening
};
web.LoadFinished += delegate {
    // stop ActivityIndicator
};
web.LoadError += delegate {
    UIAlertView error message
};
```

- To move back and forward, the following methods can be used:
 - web.GoBack()
 - web.GoForward()

These methods have a simple boolean test

- web.CanGoBack
- web.CanGoForward

- For refreshing a webpage, the following method can be used:

```
web.Reload();
```

- It is also possible to include zoom support and fitting the page

```
web.ScalesPageToFit = true;
```


MapView



These are the iOS maps, not Google Maps (Apple moved from using Google to their own maps with iOS 6).

Maps under iOS require the use of `CoreLocations` and `MapKit`. Mapping and location services are dealt with later in *Chapter 12, Peripherals*.

UIScrollView

There are times when too much information will be displayed on screen (for example, if you are dynamically generating content or creating some form of a drawing application). In such cases, `UIScrollView` can be used to ensure that the user can see everything.

The view (when combined with `PageEnabled = true;`) works by calculating the size of the page.

Assuming that the Scroll View has been created in Xcode (and is called `scrollView`), the code will be as follows:

```
private List<UIView> viewPages = new List<UIView>();
private int numPages = 4;
private float pad = 10, height =400, width = 300;
public override void ViewDidLoad()
{
    scrollView.Frame = View.Frame;
    scrollView.PagingEnabled = true;
    scrollView.ContentSize = new.SizeF(numPages * width + pad +
        2 * pad * (numPages - 1), View.Frame.Height);
}
```

We now have an effective page system. The problem is tracing which page the user is on. This is handled with `UIPageControl`. In the previous example, tracking would be performed as follows:

```
private UIPageControl pageNumber;
```

In the `ViewDidLoad()` method, tracking would be performed as follows:

```
scrollView.Scrolled += delegate {
    pageNumber.CurrentPage = (int)Math.Round(scrollView.X / width);
};
```

AdBannerView

You have seen these on many different apps. These are the bars at the top that advertise anything, from cars to fast food, and normally are targeted on the app type (for example, if you design an application that gives statistics on a car performance, the ad banner will typically get adverts for car magazines, car games, and so on). It is simply a way of generating income for the application developer. Advert support is a part of the `AdSupport` namespace.

The simple way to consider these views is as a form of web view with a number of key events:

- `AdLoaded`: until the ad is loaded, the view isn't shown. This makes the experience less obtrusive for the user.
- `FailedToReceive`: the advert failed to download.

Implementing a view with multiple View Controllers

It is simple enough to implement a view with more than one View Controller. Say we have two views. One occupies the top 130 pixels of the screen; the other is 250 pixels in height. It is added by adding the second view as a subview to the first.

```
secondView v2 = new secondView();
v2.Frame = new RectangleF(new PointF(0, 130), new
    SizeF(320, 250));
View.AddSubview(v2);
```

The events from both the View Controllers will still work as they do normally (so say the second view is a web view and the first has some buttons, the buttons will still respond to the the Touch events and the web view will still respond to the Web events).

The issue, though, comes when view 1 (the parent) wants to act on the events of view 2. This too is not that difficult to do. In fact, when dealing with the `MT.D` class in *Chapter 2, The User Interface*, the `Subview` event was acted on in the parent by overriding the `Selected` event with a delegate in another class (as demonstrated in the following code examples).

```
public class PickerChangedEventArgs : EventArgs
{
    public string SelectedValue { get; set; }
}
```

And within the class handling `UIPickerViewModel`, the `Selected` method has to be overridden to support `PickerChangedEventArgs` as shown in the following code:

```
public override void Selected(UIPickerView picker, int row,
    int component)
{
    if (PickerChanged != null)
    {
        PickerChanged(this, new PickerChangedEventArgs {
            SelectedValue = myValues[row] });
    }
}
```

Finally within the main app, code that calls the class `UIPickerView` needs to fire on the new event.

```
myPickerViewModel.PickerChanged += (object sender,
    PickerChangedEventArgs e) =>
{
    myElement.Value = e.SelectedValue;
};
```

Summary

It is a little wonder that the iPhone is such an adaptable device for displaying various forms of data with all of these Views and View types available, despite its complexity; it is actually a simple enough system to code for.

In the next chapter, we will delve deeper into controllers.

4

Controllers

In iOS app development there has to be some form of a controller, as the iOS app development framework implements the **Model View Controller (MVC)** design pattern. Controllers in iOS are split into one of the two categories: tables or not tables. While I have touched upon the `UITableView` methods when I discussed `MonoTouch.Dialog`, I've not actually given that much detail about them, which, given their importance with iOS, is rather disingenuous to say the least!

In this chapter, we will cover the following topics:

- `UITableView` and `UITableViewCell`
- `UINavigationController`
- `UITabBar`
- `UIPageControl`
- `GLKit View Controller`

UITableView and UITableViewCell

A `UITableView` method is the main workhorse for iOS. If you consider the likes of Facebook, e-mail, messages, Twitter, and many other applications, they display their data through a list-based interface. It may contain different attributes (such as images, header text, date and time, colors, and a range of other widgets), but at the end of the day, it's a table of data.

When setting up a table, there are two formats it can take: *plain* and *grouped*. The grouped view is depicted in the following image:



The plain view can be viewed as follows:



The grouped approach is better suited to username/password or settings style information, while the plain view is best for purely list-based information (such as city names, chemical elements, or tweets).

As the table requires data to propagate the view, it is usual for a delegate (or source) class to be defined as well.

Each of the elements in the preceding screenshot (such as **Brea**, **Burlingame**, and **Canoga Park**) uses `UITableViewCell`. The data can be entered or edited in these cells.

Creating a read-only table

Within Xcode, I've created a table, and into this I created another table and added a number of English Premiership teams. The simplest way to do this is through a list.

```
private List<string> premTeams;

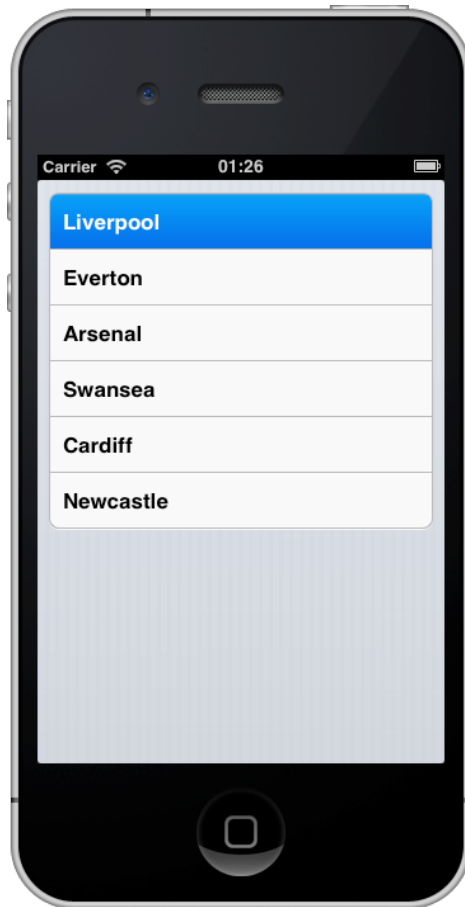
public override void ViewDidLoad() {
    base.ViewDidLoad();
    premTeams = new List<string>() {"Liverpool", "Everton",
        "Arsenal", "Swansea", "Cardiff", "Newcastle"};
    tableView.Source = new myViewSource(premTeams);
}

private class myViewSource : UITableViewSource {
    private List<string> dupPremTeams;
    public myViewSource(List<string>prems) {
        dupPremTeams = prems;
    }

    public override int RowsInSection(UITableView table,
        int section) {
        return dupPremTeams.Count;
    }

    public override UITableViewCell GetCell(UITableView tableView,
        NSIndexPath index) {
        UITableViewCell theCell = new UITableViewCell();
        theCell.TextLabel.Text = dupPremTeams[index.Row];
        return theCell;
    }
}
```

This simple code results in the following image. All that the code has done is taken the view created in Xcode and displayed the data. This is the simplest form the table can take:



Liverpool has been selected (in blue), but we're not doing anything with it. The beauty of the `UITableView` class is the power behind it. When a cell has been selected, it can be used to navigate elsewhere, have data entered in it, or just simply put a checkmark next to the selected cell value!

To enable the code to react to a cell being selected, another override has to be added to `myViewSource`, namely:

```
public override void RowSelected(UITableView tv,
    NSIndexPath index)
{ }
```

On top of `RowSelected`, the `UITableView` control provides a `DetailDisclosureButton`—this button allows multiple actions to occur within a cell. (So let's say that we have a periodic table of elements with a picture and a book icon next to the name: clicking on the name gives the atomic details and a picture of what the element looks like, and the book icon gives the history.) To enable this within the `GetCell` method, consider the following line of code:

```
theCell.Accessory = UITableViewAccessory.DetailDisclosureButton;
```

Other options available for `UITableViewAccessory` are `Checkmark` (single or multiselection within a table), `Disclosure` (a gray arrow indicating that touching the cell results in navigation), and `DetailDisclosure` (a white arrow). `UITableViewAccessory` also has a `None` option that has a placement position but nothing in it.

UITableViewCell

`TableViewCell` is plain to look at by default. As with everything in iOS, the cells can be altered in a number of ways. Cells always start off as `UITableViewCellStyleDefault`. This supports a `TextLabel` component with an optional image on its left side. The other styles are:

UITableViewCellStyle	What it does
Subtitle	Gives two left-aligned fields: <code>TextLabel</code> and <code>DetailTextLabel</code> . An image can be optionally added to the left of both of these. <code>DetailTextLabel</code> is in gray and has a smaller font size than <code>TextLabel</code> .
Value1	In this, <code>TextLabel</code> is right-aligned and blue. <code>DetailTextLabel</code> is black and left-aligned. No optional images are available.
Value2	In this, <code>TextLabel</code> is left-aligned and black. <code>DetailTextLabel</code> is right-aligned and blue. No optional images are available.

Reusable cells within a table

As with `CollectionView`, the `TableView` methods also reuse cells. The reuse is simply to prevent the app from running out of memory and slowing down. To this end, the default is for only 10 cells to be displayed at any one time (though this number may vary depending on the cells and the `UITableView` height).

To enable this reuse, the cell in my example needs to be instantiated using a different one of the overloads.

```
UITableViewCell theCell = new UITableViewCell(
    UITableViewStyle.Default, "reuseID");
```

In the preceding code, `reuseID` is the identifier that the table uses to identify the cell to be reused.

To fully demonstrate this, I have extended the list of teams in our original example and amended the `GetCell` method. All other code remains the same.

```
premTeams=newList<string>() {"Arsenal", "Aston Villa",
    "Cardiff City", "Chelsea","Crystal Palace", "Everton",
    "Fulham", "Hull City", "Liverpool", "Man City", "Man United",
    "Newcastle", "Norwich", "Southampton", "Stoke City",
    "Sunderland", "Swansea", "Tottenham", "WBA", "West Ham"
};

private class myViewSource : UITableViewSource {
    public override UITableViewCell GetCell(UITableView tableView,
        NSIndexPath index) {
        UITableViewCell theCell = tableView.DequeueReusableCell(
            "reuseID");
        if (theCell == null) {
            theCell = new UITableViewCell(
                UITableViewCellStyle.Default, "reuseID");
            // nothing to reuse, so create a cell
        }
        theCell.TextLabel.Text = dupPremTeams[index.Row];
        return theCell;
    }
}
```

Sections and Rows

Grouped data is made simpler to handle with `UITableView` as they use a `Section` and `Row` system to identify cells (these are the two parameters of `NSIndexPath`).

Indexes on a TableView

By overriding `SectionIndexTitles`, an index table can be added to the right side of a table. Clicking on an item on the index table will jump the table to that value.

```
public override string[] SectionIndexTitles(UITableView tableView)
{
    // add the index labels you want [letters are good]
    return sectionTitles.ToArray();
}
```

Customizing a `UITableViewCell` component can be performed in or out of Xcode. To be honest, the simplest way is doing it in Xcode. Drag a cell to an empty view and add the widgets you want in there.

Navigation with UITableView

Navigation with tables is provided by `UINavigationController`. Simply put, `UINavigationController` allows you to move between table views with the standard back button at the top of the screen. The navigation controller can be set up using Xcode or purely with code.

Within code

Setting up the `UINavigationController` is performed within the `AppDelegate.cs` file.

```
UIWindow window = new UIWindow(UIScreen.MainScreen.Bounds);
var viewController = new mainController();
UINavigationController rootNavigationController =
    new UINavigationController();
rootNavigationController.PushViewController(viewController,
    false);
window.RootViewController = rootNavigationController;
window.MakeKeyAndVisible();
```

This places the navigation bar at the top of the screen. When a new view is shown, the back button becomes visible. The back button can be hidden using the following line of code:

```
NavigationItem.HidesBackButton = true;
```

The `UINavigationController` can also be hidden using the following:

```
NavigationController.NavigationBarHidden = true;
```

The problem here is that as you go forward to a view and back again, the bar will no longer be hidden. The code that has to be called is the `ViewWillAppear` method.

With Xcode

There are a couple of ways of using `NavigationController` within Xcode. However, they can be quite painful to use.



Image A

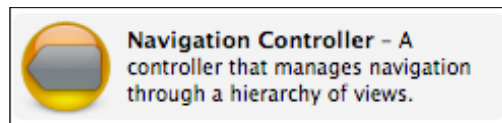


Image B

Navigation Controller (Image B) creates a view with the controller bar at the top. It has the big advantage of having both the navigation section and an area for a view controller to be loaded into (Image A). To use this, simply create a view controller as you would. Within the `RootViewController` source, the following will load the other `ViewController` into the `RootViewController`:

```
var myViewController = new MyViewController();
myViewController = "My View";
// sets the title and gives a back button

NavigationController.PushViewController(myViewController, true);
```

Each navigation controller requires a bar with an optional navigation controller button.



The title bar can also be placed into a typical view and connected in the standard way. **Navigation Item** is not what it seems. By the looks of the item, it would be presumed that the button can be used for the likes of a back button, menu button, or the buttons that can be placed on the title bar. *It isn't!* For that, `BarButtonItem` has to be added to the title bar directly and the **Navigation Item** should be connected to the bar, which can be seen as follows:



Image C

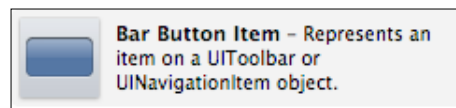
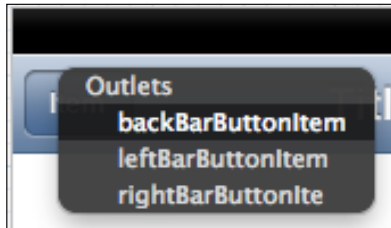
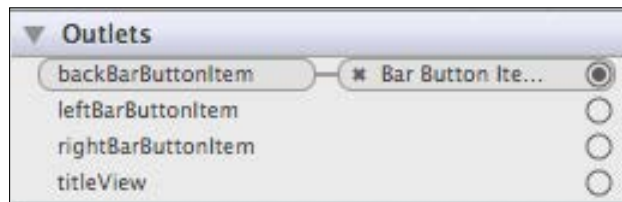


Image D

Drag the **Navigation Item** (Image D) to the bar so that it is under the View icon (Image C). Drag the **Bar Button Item** onto a title bar already on the view. This will give you a bar with a button on the left (assuming you placed it on the left). The next stage is to connect the **Navigation Item** to the button on the bar. To do this, press the *Shift* button and drag the **Navigation Item** onto the **Bar Button Item**. When that has been done, you will see a menu giving you the option of what that item represents, shown as follows:



Selecting **backBarButtonItem** will give the back button. If you select **leftBarButtonItem** or **rightBarButtonItem**, these will give a button that can be used for other purposes (such as icons for an address book, or menu). To check whether the connection has been made, the outlet's connection on the right will confirm it:



To access the `backBarButtonItem` within the code, something akin to the following can be used:

```
NavigationItem.BackBarButtonItem.Clicked += delegate {  
    UINavigationController.PopViewControllerAnimated(true);  
    // sends back to the previous view on the stack  
};
```

Navigation using UITableView

At this point, I will assume that the view already has a valid table with some data in it. One of the key points of the `UITableView` is that the user can select a table cell and that cell can be moved to another view.

For the following line of code:

```
RowSelected(UITableView tableView, NSIndexPath indexPath)
```

To start a new `ViewController`, we need to do the following:

```
var newController = new myNextTableViewController(ref_to_data,
    rows[indexPath.Row]);
newController.Title = rows[indexPath.Row];
parent_controller.NavigationController.PushViewController(
    myController, true);
tableView.DeselectRow(indexPath, true);
```

This will start a new view, and when the back button is selected the view reappears. The **backBarButtonItem** title will show the title of the view that started the new view. `DeselectRow` removes the blue selection color from the view. If it is not called when the back button is pressed, the selection will still show as being selected.

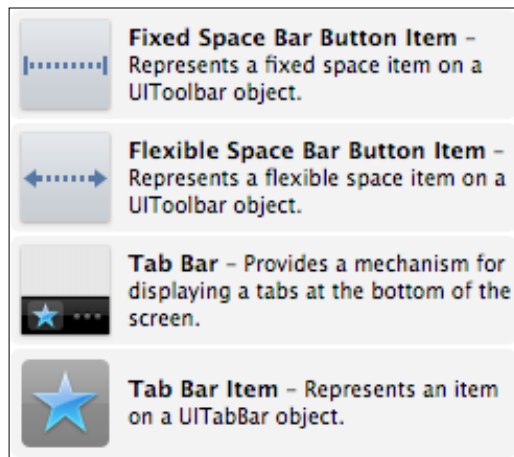
Returning to the RootView

To return to the top level of the view, use the following method:

```
NavigationController.PopToRootViewController(true);
```

TabBars

Another simple method of navigation is through `UITabBar`. Typically, the bars are at the bottom of the screen, but there is nothing to stop a view having a `UITabBar` at the top of the screen as well.



Placing a tab bar on a view is the same as placing any other view onto a screen; drag it onto the screen. By default, two `TabBarItem` items are added to `TabBar`. You can add more by dragging the **Tab Bar Item** widget onto the bar.

Each `TabBarItem` has a `Title` and `Image` property that can be set; however, these buttons must be set as **Custom** type within Xcode.

The two **Space Bar Button Item** components allow for button items to be added to a view with space between them. **Fixed** has a finite amount of space between buttons, while **Flexible** allows more buttons to be added but for them to be kept at equidistant.

The `TabBar` components are not the same as the other mechanisms for navigation. While a `TabBarItem` can certainly start a new view, it is more common that the same view is used all the time but the new view controller is loaded into the blank frame between the navigation bar (or whatever you have at the top of the screen and the `TabBarItem` at the bottom).

The `TabBarItem` components are also good if a view containing the `TabBar` is generating the view dynamically and displaying on screen.

Handling the Tab Bar in code

Within your code, it is not the **Tab Bar Item** that needs to be handled with events but the **Tab Bar** itself. The **Tab Bar Item** components will still need connecting, so they can be handled within the main controller source.

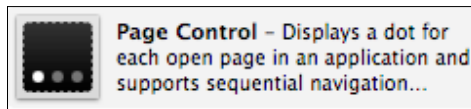
```
tabBar.ItemSelected += TabBarSelected;
```

Possibly the simplest method of determining which **Tab Bar Item** has been selected is to set the `Tag` property on each (either in Xcode or in controller source). In `TabBarSelected`, the correct `TabBarItem` can then be found.

```
private void TabBarSelected(object s, EventArgs e)
{
    UITabBar item = (UITabBar)s;
    UITabBarItem item = tb.SelectedItem;
    switch(item.Tag)
    {
        // do something;
    }
}
```

PageControl

The `UIPageControl` method is a handy mechanism to mark that we have multiple pages on a single view. It is typically used with a `UIScrollView` to display the index of the page in `UIScrollView`.



They can also be used for navigation. When you click on the left or right side of the page, an event is thrown for page movement. The number of dots indicates the number of pages. This number can always be altered in the code.

GLKit

A `GLKit` view enables the use of **OpenGL for Embedded Systems (OpenGL ES)** in an application. Animation and graphics are outside the scope of this book, but there are a large number of Xamarin.iOS examples available that can demonstrate how to use OpenGL ES.

Summary

This chapter has given you the basics of the two most used forms of *Navigation* and *View*. While `UITableView` is undoubtedly extremely powerful, it is cumbersome when you have the ability to use the much simpler `MonoTouch.Dialog` class.

5

UI Controls

To paraphrase Edmund Blackadder: "A UI without a control is like a pencil without a nib – pointless."

The range of controls, while not that large, is very flexible in what you can do with them. All controls can be created within Xcode or dynamically within the code.

In this chapter we will be covering the following topics:




















- Controls and widgets
- Control selection
- Control customization
- Control reference (Android and iOS cross reference)

Controls and widgets

When we refer to a widget, I use it interchangeably with a UI Control. In old terms, a widget stands for a **W**in**D**ow **g**ad**G**ET. The screen on an iOS device is classed as a window, so it's fair enough to call anything on screen a widget.

UI Controls

The UI Controls are directly available from Xcode. For creating within code, the control name is preceded with UI and all/any spaces removed (except **Round Rect Button** which is just `UIButton`). The **Fixed Space Bar Button Item** and **Flexible Space Bar Button Item** are accessed from `UIBarButtonItem`.

 Label – A variably sized amount of static text.	 Navigation Bar – Provides a mechanism for displaying a navigation bar just below the status...
 Round Rect Button – Intercepts touch events and sends an action message to a target object when...	 Navigation Item – Represents a state of the navigation bar, including a title.
 Segmented Control – Displays multiple segments, each of which functions as a discrete button.	 Search Bar – Displays an editable search bar, containing the search icon, that sends an action message...
 Text Field – Displays editable text and sends an action message to a target object when Return is tapped.	 Search Bar and Search Display Controller – Displays an editable search bar connected to a search...
 Slider – Displays a continuous range of values and allows the selection of a single value.	 Toolbar – Provides a mechanism for displaying a toolbar at the bottom of the screen.
 Switch – Displays an element showing the boolean state of a value. Allows tapping the control to...	 Bar Button Item – Represents an item on a <code>UIToolbar</code> or <code>UINavigationController</code> object.
 Page Control – Displays a dot for each open page in an application and supports sequential navigation...	 Fixed Space Bar Button Item – Represents a fixed space item on a <code>UIToolbar</code> object.
 Stepper – Provides a user interface for incrementing or decrementing a value.	 Flexible Space Bar Button Item – Represents a flexible space item on a <code>UIToolbar</code> object.
 Date Picker – Displays multiple rotating wheels to allow users to select dates and times.	 Tab Bar – Provides a mechanism for displaying a tabs at the bottom of the screen.
	 Tab Bar Item – Represents an item on a <code>UITabBar</code> object.

Control selection

Most of the controls shown in the previous screenshot are obvious for what they do, and are easy enough to just add and connect to the UI. However, there are some controls that need to be considered differently from the others, most notably `UIButton` and `UIStepper`. The `UIButton` class is very flexible in what it can do, while `UIStepper` really needs to be used as an **action** rather than an **outlet**. This means they have to be handled in a different way.

UIButton

A `UIImageView` has no events attached to it. So, if you need an image that can be clicked on, they can be planted onto a `UIButton` class.

```
var r = new UIButton();
r.ImageView = UIImage.FromFile("path/toimage.png");
```

It is completely possible to add text as well as an image to the button by adding a `UILabel` to the button, but it is a two step process. The important thing to remember is that, when placing anything on another view, the size of the parent has to be taken into consideration, as shown in the following code:

```
var r = new UIButton();
r.Frame = new CGRect(0, 0, 100f, 100f);
// button 100x100 at 0, 0
var i = new UIImageView(new CGRect(15f, 2f, 70f, 70f));
i.Image = UIImage.FromFile("path/toimage.png")
    .Scale(new CGSize(70f, 70f));
var l = new UILabel(new CGRect(2f, 78f, 96f, 20f));
l.Text = "Hello world";
r.AddSubview(i);
r.AddSubview(l);
```

You can add as many images and labels to a button. The button can also have a background image added to it, as shown in the following code:

```
r.SetBackgroundImage(UIImage.FromFile("path/toimage.png"),
    UIControlState.Normal);
```

Alternatively, the background can have a gradient fill added, as shown in the following code:

```
var gradient = new CAGradientLayer();
gradient.Colors = new MonoTouch.CoreGraphics.CGColor[]
{
    UIColor.FromRGB(115, 181, 216).CGColor,
    UIColor.FromRGB(35, 101, 136).CGColor
};
gradient.Locations = new NSNumber[]
{
    .5f,
    1f
};
gradient.Frame = r.Layer.Bounds;
r.Layer.AddSublayer(gradient);
r.Layer.MasksToBounds = true;
```

Say, the `UILabel` class we added before was too small for the text coming in (it's no longer `Hello world, but I love drinking hot coffee`), the font size within the label will need to be changed. The resizing is performed by creating a bounding box (`textSize`), and fitting the text inside of it by setting the font size so that the text fits the height – not the width – of the bounding box.

```
RectangleF textSize = new RectangleF(i.X, i.Y, i.Width, i.Height);  
l.Font = UIFont.SystemFontOfSize(textSize.Height);  
l.Text = "I love drinking hot coffee";
```

The button doesn't have to have a rounded shape – a custom shape can be also applied. Let's look at the following code for a circular button:

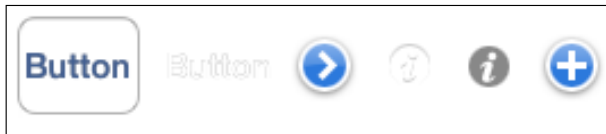
```
r.Frame = CGPath.EllipseFromRect(new RectangleF  
    (135f, 180f, 40f, 40f));  
//width and height should be same value  
r.ClipsToBounds = true;  
r.Layer.CornerRadius = 20; //half of the width  
r.Layer.BorderColor = UIColor.Red.CGColor;  
r.Layer.BorderWidth = 2.0f;
```

For this to work, the `Monotouch.CoreGraphics` namespace has to be included.

There are other types of buttons also available (`ContactAdd`, `Custom`, `DetailDisclosure`, `InfoDark`, `InfoLight`, `RoundedRect`, and `System`). While `RoundedRect` is the most common form used, the others can be created either in Xcode or as code in your app.

```
var myButton = new UIButton(UIButtonType.Custom);
```

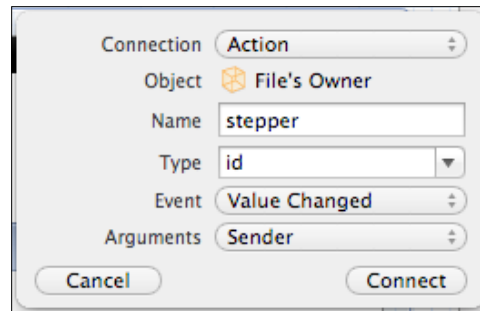
The preceding code creates a button of type `Custom`. The other types of buttons look as shown in the following screenshot:



From left to right (System, Custom, DetailDisclosure, InfoLight, InfoDark, ContactAdd)

UIStepper

The `UISteppers` class needs to be defined in the connector as both an outlet and action (with the **Event** selected to be **Value Changed**). A typical action is shown in the following screenshot:



```
public override void ViewDidLoad()
{
    base.ViewDidLoad();
    uiStepper.MinimumValue = 0;
    uiStepper.MaximumValue = 10;
    uiStepper.AutoRepeat = true;
    lblCounter.Text = uiStepper.Value.ToString();
}
partial void stepper(NSObject sender)
{
    UIStepper step = (UIStepper)sender;
    lblCounter.Text = step.Value.ToString();
}
```

When the Stepper control is clicked on, the `partial void` method is called, which updates the counter as shown in the preceding code. If you just have one without the other, the app will die when it comes to a View Controller with a `UIStepper` class on.

The other controls

The following table will give you an indicative list of what each control does, and any caveats for the use of the control:

Control	Used for	Caveats
Label	Simple labels	Does not respond to touch events
SegmentedControl	A multiple button on a single view. Frequently used on MapViews for the different types of map available	
TextField	Text entry	Set <code>SecureTextEntry = true</code> for passwords
Slider	Selecting a volume or color	Uses float values
PageControl	A simple method of showing how many pages are on a scrollview area	
DatePicker	A date picker	Can also be used for Time as well
NavigationBar	A navigation bar for the top of the screen	Requires a navigation item
NavigationItem	A navigation item for the navigation bar	Needs a Navigation Controller to be added for the item type to be defined within Xcode
SearchBar	A search bar	Needs both an action and outlet defined
ToolBar	A bar used for adding buttons to, as tools	Use this for the event and in the handler use <code>Item</code> for the <code>BarButtonItem</code> .
BarButtonItem	A button for use on a toolbar. Comes with an image and label already on the button	Don't catch the event from this; use the toolbar

Control	Used for	Caveats
Fixed/ FlexibleBarButtonItem	Provides space between buttons on the toolbar	
TabBar	Used for tabviews	Use a Toolbar controller to define which nib file, pressing a tab, will call
TabBarItem	A button for use on a TabBar, similar to aBarButtonItem	

Comparing Android to iOS UI controls

The following table is a comparison list and is intended for those wishing to port to or from Android:

Android	iOS	Android responds to events	iOS responds to events
Button	UIButton	Yes	Yes
Text	UILabel	Yes	No
ListView	UITableView	Yes	Yes
	UITableViewCell		Yes
CheckBox	Switch	Yes	Yes
CheckedTextView	Switch with UILabel	Yes	Yes
ProgressBar	ActivityIndicatorView	Yes	No
RadioButton	MTD.RadioElement	Yes	Yes
RadioGroup	MTD.RadioGroup	Yes	No
SeekBar	Slider	Yes	Yes
TextView (multiple types)	TextField	Yes	Yes
FrameLayout, LinearLayout, RelativeLayout, TableLayout		Yes	
GridView	ComponentView	Yes	Yes
ScrollView	ScrollView	Yes	Yes
SlidingDrawer	ActionBar	Yes	

Android	iOS	Android responds to events	iOS responds to events
TabHost	TabBar	No	Yes
TabWidget	TabBarItem	Yes	No
WebView	Webview	Yes	Yes
Gallery	ImageGallery	Yes	Yes
ImageButton	Button	Yes	Yes
ImageView	ImageView	Yes	No
MediaController	AudioView Controller	Yes	Yes
VideoView	VideoView Controller	Yes	Yes
DatePicker	DatePicker	Yes	Yes
TimePicker	DatePicker	Yes	Yes
DialerFilter	UIKeyboard	Yes	Yes
GestureOverlayView	Gestures (approx)	Yes	Yes
SurfaceView	View	Yes	Yes
TwoLineListItem	MTD.MultipleLineElement	Yes	Yes
View	View	Yes	Yes
Zoom Button	Stepper	Yes	Yes
Zoom Controls	SegmentedControl	Yes	Yes

Although there are other views/controls on Android that are not on iOS and vice versa, these are the most common ones.

Summary

With the wealth of UI Controls to play around with and the degree of customization that can be achieved on just about everything, it is little wonder that people enjoy using their iOS devices; they can be made to look good!

6 Events

Events are an essential aspect of any iOS application. In fact, without them, your phone will just sit there like a useless lump of plastic! Every time something happens, an event is raised. For the developer, events are everything.

In this chapter we will be covering the following topics:

- Handling events
- Event reference
- Control event reference

Handling events

Events are handled using one of the following types:

- Delegates
- Handlers

Delegates

You can think of a delegate event as an inline event. They can be anonymous or can use the event as follows:

- When the delegate is anonymous, the code will be as follows:

```
var uiButton = new UIButton();
uiButton.TouchUpInside += delegate {...};
```
- When the delegate is using an event, the code will be as follows:

```
var uiButton = new UIButton();
uiButton.TouchUpInside +=
    (object sender, EventArgs e) => {...};
```

If the button doesn't have anything that requires either the sender or the event, the anonymous event can be used.

Xamarin.iOS delegates all Events to use `EventArgs e` when not using a handler.

Attaching an event to multiple controls

For arguments sake, I will assume that a View Controller has four buttons. There is nothing wrong with assigning each button its own event handler, but it is a waste of memory if one event can handle multiple events. For sanity reasons though, this should be restricted so that the `TouchUpInside` events are handled together rather than everything on a view (or View Controller) that accepts the `TouchUpInside` event.

A simple solution is to have a single event for all of the buttons. However, the issue is how to recognize and act on the correct button in the event handler itself.

Possibly the simplest method is to set the `Tag` property with a number, as shown in the following code:

```
private void setup()
{
    UIButton btn1 = new UIButton()
    {
        // set up the properties
        Tag = 1,
    };
    UIButton btn2 = new UIButton()
    {
        // set up the properties
        Tag = 2,
    };
    UIButton btn3 = new UIButton()
    {
        // set up the properties
        Tag = 3,
    };
    btn1.TouchUpInside += HandleButtonPressedEvent;
    btn2.TouchUpInside += HandleButtonPressedEvent;
    btn3.TouchUpInside += HandleButtonPressedEvent;
}
```

```
private void HandleButtonPressedEvent(object sender, EventArgs e)
{
    UIButton theButton = (UIButton)sender;
    switch(theButton.Tag)
    {
        // do what is needed
    }
}
```

Synchronous versus asynchronous event handling

Events come in two flow methods: synchronous and asynchronous. The difference between them is as important as understanding their uses. To understand the differences, you need to think of two people taking a walk.

Synchronous walk

The two people walk together and come to a traffic light at a road; both stop, and when the traffic light turns green they set off together. At the end of the walk, they are still together and sit down for a beer in the sun.

Asynchronous walk

The two people start off together to the pub but when they come to the traffic light, one of the two stops to wait for the signal while the other crosses. Once the signal allows a safe crossing, the one who stopped walks across and carries on. The one who just crossed has been stopped by the police for jaywalking. There is no guarantee that the pair will reach the pub at the same time; it depends on other factors slowing them down.

In a programming context

Consider a simple messenger application, it consists of two parts: grabbing the messages and displaying them. The synchronous event would be for calling the address View Controller. Here, clicking on a button can be thought of as two people walking together and coming to a stop.

The asynchronous part is the downloading of the messages; here, the slow part is grabbing the messages from the server. If this was to be performed by a synchronous event, the messages would be requested and the thread would be frozen while a return request from the server was obtained. While that happens the UI is created, but how can the UI be created with no data? In a nutshell, it can't and the app dies. As it is being performed in an asynchronous way, the creation of the UI doesn't happen until the server has sent everything. When that happens, the next stage can be performed.

For example, consider the following code (this has been taken directly from the Xamarin ZXing component and is released through the creative commons license):

```
buttonCustomScan.TouchUpInside += async (sender, e) =>
{
    //Create an instance of our custom overlay
    customOverlay = new CustomOverlayView();
    //Wireup the buttons from our custom overlay
    customOverlay.ButtonTorch.TouchUpInside += delegate {
        scanner.ToggleTorch();
    };
    customOverlay.ButtonCancel.TouchUpInside += delegate
    {
        scanner.Cancel();
    };

    //Tell our scanner to use our custom overlay
    scanner.UseCustomOverlay = true;
    scanner.CustomOverlay = customOverlay;
    var result = await scanner.Scan();
    HandleScanResult(result);
};
```

The important part of the code is the line `var result = await scanner.Scan();` until that has returned, the `HandleScanResult` method will not be called. The `TouchUpInside` events within the `async` handler are synchronous and once clicked on, they either terminate the handler or start another piece of code – they both act immediately.

Asynchronous calls are supported by Xamarin.iOS (and Xamarin.Android) due to the support for it within Mono 3.

Events and controls reference

Not every type of widget has events attached to it by default (there is nothing to stop you from writing an event and adding it to a widget though), for example, `UIImageView` doesn't have anything attached.

The iOS makes a big play over its touch system, which is reflected in the number of events given over to touches. Unless the widget has no events attached to it, the following table applies to all the widgets:

Name of event	What it does and when
<code>TouchCancel</code>	It is called when cancelling the current touches for a control (any control that handles the <code>TouchCancel</code> event).
<code>TouchDown</code>	This is the touch-down event on a widget.
<code>TouchDownRepeat</code>	This is a repeated touch-down event. For this event to be triggered the <code>UITouch tapCount</code> value must be >1 .
<code>TouchDragEnter</code>	It is called when a finger is dragged into the bounds of the control.
<code>TouchDragExit</code>	It is called when a finger is dragged outside the bounds of the control.
<code>TouchDragInside</code>	It is called when a finger is dragged inside the bounds of the control.
<code>TouchDragOutside</code>	It is called when a finger is dragged just outside the bounds of the control.
<code>TouchUpInside</code>	It is called when a finger is inside the bounds of the control.
<code>TouchUpOutside</code>	It is called when the finger is outside the bounds of the control.

Other significant control events

The events listed in the following table are found on many of the controls in iOS:

Name of event	What it does and when
<code>ValueChanged</code>	It allows access to the changed value on a control when the value of the control has changed (for example, a <code>UITextField</code> having the text altered).
<code>EditingDidBegin</code>	It is a touch starting an edit session within a <code>UITextField</code> class.
<code>EditingChanged</code>	It is emitted when the value of a <code>UITextField</code> class has changed.

Name of event	What it does and when
EditingDidEnd	It is a touch ending an edit session of a UITextField class by leaving the bounds.
EditingDidEndOnExit	It is a touch ending an edit session of a UITextField class.
AllTouchEvent	It intercepts all touch events.
AllEditingEvents	It intercepts all edit events.
AllEvents	It intercepts all events (including system events).

TouchesBegan, Moved, Ended, and Cancelled as well as the gesture recognizer events are covered in the next chapter.

AVAudioPlayer and AVRecordClass

Playing and recording audio and video is a key feature of the iOS experience. The following table lists the events you will need to hook onto for playing and recording to go smoothly:

Name of event	What it does and when
BeginInterruption	It is raised when the audio player is interrupted (say by a phone call).
DecodeError	It is raised when a file cannot be decoded.
EndInterruption	It is raised when an interruption has finished.
FinishedPlaying	It is raised when a file has finished playing.

AVAudioSession

The audio session is the physical act of playing a piece of audio or video. A number of events can interrupt a video or audio that is being played. The following table lists the events you need to know:

Name of event	What it does and when
BeginInterruption	It is raised when an audio session is interrupted.
CategoryChanged	It is raised when an audio session category is changed.
EndInterruption	It is raised when an interruption to the audio session is completed.
InputAvailabilityChanged	It is raised when the availability of an audio input changes on a device.
InputChannelsChanged	It is raised when the number of input channels are changed.

Name of event	What it does and when
OutputChannelsChanged	It is raised when the number of output channels are changed.
SampleRateChanged	It is raised when the sample rate is altered.

ABAddressBook

The address book within the iOS can be altered from both the address book facility and from outside of the facility.

Name of event	What it does and when
ExternalChange	It is raised when something external to the address book (such as a user application that alters the address book) has changed something within the address book.

ABNewPersonViewController

As with all the views, a View Controller is there for the UI to be placed on. The address book `NewPersonViewController` is no different.

Name of event	What it does and when
NewPersonComplete	It is called when the user clicks on the Save or Cancel button. If Save is clicked on, the new contact is saved to the address book database.

ABPeoplePickerNavigationController

The `ABPeoplePickerNavigationController` acts in the same way as a normal navigation controller, but with a couple of extras added in, as shown in the following table:

Name of event	What it does and when
Cancelled	It is sent when the Cancel button is clicked on.
PerformAction	It is raised when an object on the person picker is selected.
SelectPerson	It is raised when a person is selected.

ABPersonViewController

The **Person View Controller** displays the selected person with a single event attached.

Name of event	What it does and when
PerformDefaultAction	It is sent when the user selects a property value for a person in the Person View Controller.

ABUnknownPersonViewController

Unknown Person View Controllers display the information about a person prior to being accepted into the **ABAddressBook** class. Its events are listed on the following table:

Name of event	What it does and when
PerformDefaultAction	It is sent when the user selects a property value for a person in the Person View Controller.
PersonCreated	It is called when the user clicks on the Save or Cancel button. If Save is clicked on, the new contact is saved to the address book database.

AudioConverter

The **AudioConverter** class is used for converting audio formats.

Name of event	What it does and when
InputData	It is raised when data is being given as input through a port.

AudioSession

The **AudioSession** class is similar to the **AVAudioSession** class but it is exclusively for audio.

Name of event	What it does and when
AudioRouteChanged	It is raised when the route for the output changes (for example, speaker to headphones).
Interrupted	It is raised when the audio session is interrupted.
Resumed	It is raised when the interruption to the audio session has completed.

InputAudioQueue

The input queue is used when audio is being fed into the device.

Name of event	What it does and when
InputCompleted	It is raised when the input has completed.

OutputAudioQueue

The output queue is for audio output.

Name of event	What it does and when
OutputCompleted	It is raised when the audio output has completed.

AUGraph and AudioUnit

Both `AUGraph` and `AudioUnit` share this event, which has the same effect for both.

Name of event	What it does and when
RenderCallback	It is the callback used for rendering during audio graphing.

AudioConverter

The callback event is used to callback an object to/from an event.

Name of event	What it does and when
EncoderCallback	It is the callback created for the converter encoder.

CAAnimation

Animation has two key events; start and end.

Name of event	What it does and when
AnimationStarted	It is called when the animation starts.
AnimationEnded	It is called when the animation ends.

CBCentralManager

The `CoreBluetoothCentralManager` is the class that handles the adding or removing of bluetooth devices. The handling events have been listed in the following table:

Name of event	What it does and when
<code>ConnectedPeripheral</code>	It is raised when a connection is successfully made.
<code>DisconnectedPeripheral</code>	It is raised when a connection is successfully disconnected.
<code>DiscoveredPeripheral</code>	It is raised when a connection is discovered.
<code>FailedToConnectPeripheral</code>	It is raised when a connection to a peripheral fails.
<code>RetrievedConnectedPeripheral</code>	It is raised after a connected peripheral information is retrieved.
<code>RetrievedPeripherals</code>	It is raised after requesting a list of all stored peripherals.
<code>UpdatedState</code>	It is called after the state of a connection changes.

CBPeripheral

Each bluetooth device has characteristics and descriptors attached to them. The events have been listed in the following table:

Name of event	What it does and when
<code>DiscoverCharacteristic</code>	It discovers the characteristics of a service for the peripheral.
<code>DiscoveredDescriptor</code>	It discovers the descriptors of a characteristic.
<code>DiscoveredIncludedService</code>	It discovers the specified included services for the peripheral.
<code>DiscoveredService</code>	It is called after discovery of the specified service for the peripheral has completed the connection.
<code>InvalidatedService</code>	It is raised when the peripheral services change.
<code>RssiUpdated</code>	It is raised when the value for the peripheral's current RSSI while connected to the <code>CoreBluetooth</code> central manager.
<code>UpdatedCharacteristicValues</code>	It is raised after the value for a characteristic has been updated.

Name of event	What it does and when
UpdatedName	It is raised when the peripheral name is changed.
UpdatedNotificationState	It is raised when the peripheral notification state is changed.
UpdatedValue	It is raised when the peripheral characteristic descriptor value is changed.
WroteCharacteristicValue	It is raised after the value for a characteristic has been written.
WroteDescriptorValue	It is raised after the value for a descriptor has been written.

CBPeripheralManager

The `CBPeripheralManager` class manages the peripherals attached to the bluetooth manager.

Name of event	What it does and when
AdvertisingStarted	It is raised when the peripheral advertising its presence has started.
CharacteristicSubscribed	It is raised when a remote device subscribes to a characteristic's value.
CharacteristicUnsubscribed	It is raised when a remote device unsubscribes to a characteristic's value.
ReadRequestReceived	It is raised after a read request has been received.
ReadyToUpdateSubscribers	It is invoked when a local peripheral is ready to send a characteristic's updated value.
ServiceAdded	It is raised after a service has been added.
StateUpdated	It is raised after a peripheral state has been updated.
WriteRequestsReceived	It is raised after a write request has been received.

CFSocket

The `CoreFoundation` socket covers the connection to a remote socket (typically online).

Name of event	What it does and when
<code>AcceptEvent</code>	It is invoked when <code>CoreFoundation</code> is set to accept events from a socket.
<code>ConnectEvent</code>	It is raised when a client connects to a remote socket it called.

CFStream

Similar to an `IOStream`, the `CFStream` deals with data to and from a socket.

Name of event	What it does and when
<code>CanAcceptBytesEvent</code>	It is raised when the stream has information available for writing.
<code>ClosedEvent</code>	It is raised when a close operation on the stream completes.
<code>ErrorEvent</code>	It is raised when an error occurs on the stream.
<code>HasBytesAvailableEvent</code>	It is raised when the stream has information available for reading.
<code>OpenCompletedEvent</code>	It is raised when an open operation on the stream completes.

CLLocationManager

The `CoreLocation` `CLLocationManager` class is the control class for the location manager on the iOS device.

Name of event	What it does and when
<code>AuthorizationChanged</code>	It is called when the user allows or prevents the use of <code>CoreLocation</code> functions.
<code>DeferredUpdatesFinished</code>	It is raised when the deferred updates time is over.
<code>DidStartMonitoringForRegion</code>	It informs the delegate that a new region is being monitored.
<code>Failed</code>	It is raised when <code>CLLocationManager</code> fails to start.
<code>LocationUpdatesPaused</code>	It pauses updating the location.
<code>LocationUpdatesResumed</code>	It restarts paused location updates.
<code>LocationsUpdated</code>	It updates the GPS position.

Name of event	What it does and when
MonitoringFailed	It is raised when CLLocationManager fails to monitor (no communication with the satellite is a usual cause).
RegionEntered	It is called when entering a region.
RegionLeft	It is called when a region has been left.
UpdatedHeading	It is called when a heading is updated.
UpdatedLocation	It is called when the location has been updated.

MidiClient

Possibly not that useful on an iPhone (though it is on an iPad), iOS has a rich MIDI layer.

Name of event	What it does and when
IOError	It is raised when the MIDI client suffers an input/output error.
ObjectAdded	It is called when a MIDI device is added.
ObjectRemoved	It is called when a MIDI device is removed.
PropertyChanged	It is invoked when a MIDI property for a device is changed.
SerialPortOwnerChanged	It is invoked when the serial port owner has changed (effectively called when a device is switched).
SetupChanged	It is raised when the client setup has changed.
ThruConnectionsChanged	It is raised when the MIDI daisy chain connection alters.

MidiEndpoint and MidiPort

Both of these classes have the same named event and it does exactly the same in both!

Name of event	What it does and when
MessageReceived	It is raised when the MIDI subsystem receives a message from the device.

Monotouch.Dialog

MonoTouch.Dialog is an extremely powerful class that takes much of the trouble out of creating and using UITableViews in your app.

BadgeElement, BaseBooleanImageElement, GlassButton, LoadMoreElement, MessageElement, and StringElement

These classes all have the `Tapped` event in. It acts when the option is tapped. Be aware that it sends an `NSAction` event rather than a typical object sender / `EventArgs` combo, which means that you cannot use the same handler for multiple instances of a class.

Name of event	What it does and when
<code>Tapped</code>	It is emitted when the Element has been tapped.

BoolElement

This is a simple on/off element.

Name of event	What it does and when
<code>ValueChanged</code>	It is emitted when the Boolean has changed.

DateTimeElement

When invoked, a standard `DateTime Picker View` is produced.

Name of event	What it does and when
<code>DateSelected</code>	It is called when the date has been selected.

DialogViewController

The `DialogViewController` is the View Controller the `MT.D` is placed in.

Name of event	What it does and when
<code>OnSelection</code>	It is invoked when an object within a DVC is selected.
<code>RefreshRequested</code>	It is called when a refresh of the <code>MT.D</code> is called.
<code>SearchTextChanged</code>	It is called when the search Text Field text has changed.
<code>ViewAppearing</code>	It is raised when <code>MT.D</code> view is being created.
<code>ViewDisappearing</code>	It is raised when <code>MT.D</code> view is being disposed.

EntryElement

An entry element allows for data entry into a `UITextField` held within a `UITableViewCell`.

Name of event	What it does and when
Changed	It is called when the <code>EntryElement</code> content has been changed.
ShouldReturn	It asks if the Text Field should process the Return button.

StyledStringElement

The `StyledStringElement` class is the same as a `StringElement` class, except that you can add styles to it.

Name of event	What it does and when
AccessoryTapped	It is emitted when the element is tapped.

EKCalendarChooser

The `CalendarChooser` class of `EventKit` allows the user access to a calendar.

Name of event	What it does and when
Cancelled	It is called when the user cancels choosing.
Finished	It is called when the user selects Done .
SelectionChanged	It is called when the date selected is changed.

EKEventEditViewController and EKEventViewController

Both of these `EventKit` View Controllers have the same named event that has the same effect for both.

Name of event	What it does and when
Completed	It is emitted when the control has finished its action.

EAAccessory

The external accessory class deals with any accessory not part of the phone or bluetooth.

Name of event	What it does and when
Disconnected	It is issued when an external accessory has been disconnected.

The NS classes

NS stands for **NextStep**. They are a bunch of classes that formed a part of the lineage when Apple bought NeXT after Steve Jobs returned from the wilderness. While there are not that many events attached to them, they are vital. Many of the NS classes are required by the bindings between Xamarin.iOS and the Objective C underlayer.

NSCache

NSCache is an internal cache system used for many different jobs.

Name of event	What it does and when
WillEvictObject	It is called when an object is about to be removed from the cache.

NSKeyedArchiver

The KeyedArchiver class encodes data with a key.

Name of event	What it does and when
EncodedObject	It is called when an object has been encoded.
Finished	It is raised when the encoding has finished.
Finishing	It is raised when the encoding is about to finish.
ReplacingObject	It informs the delegate that a given object is going to be replaced by another object.

NSKeyedUnarchiver

The `KeyedUnarchiver` class performs the reverse of the `KeyedArchiver` class.

Name of event	What it does and when
Finished	It is raised when the decoding has finished.
Finishing	It is raised when the decoding is about to finish.
ReplacingObject	It informs the delegate that a given object is going to be replaced by another object.

NSNetService

This is the class used for network services.

Name of event	What it does and when
AddressResolved	It is emitted when the address has been resolved.
PublishFeature	It informs that a service feature was published successfully.
Published	It informs that a service was published successfully.
ResolveFailed	It is emitted when the address can't be resolved.
Stopped	It informs that a request to publish or resolve was stopped.
UpdatedTxtRecordData	It notifies that a TXT record for a service has been updated.
WillPublish	It informs that the network is ready to publish a service.
WillResolve	It informs that the network is ready to resolve a service

NSNetServiceBrowser

The `NetServiceBrowser` class is used for connection to the outside world.

Name of event	What it does and when
DomainRemoved	It informs when a domain has disappeared or is no longer available.
FoundDomain	It is raised when the sender has found a domain.
FoundService	It is raised when the sender has found a service.
NotSearched	It is raised when the search was not successful.
SearchStarted	It is raised when a search has begun.
SearchStopped	It is raised when a search has been stopped.
SearchRemoved	It is raised when a search has been removed from the browser.

NSStream

This is similar to a standard .NET Stream.

Name of event	What it does and when
OnEvent	It is called when a given event occurs on a given stream.

GLKView

The `GLKitView` is a view used for `OpenGL` graphics.

Name of event	What it does and when
DrawInRect	It draws the view's content within a given rectangle.

GK classes

Games are an important part of any user experience. It's fine if you just want a phone to be a phone, but if you have the capabilities to play Angry Hedgehogs or a football manager game, then why not use them?

GKAchievementViewController, GKFriendRequestComposeViewController, and GKLeaderboardViewController

These three classes have the same named event.

Name of event	What it does and when
DidFinish	It is called when the view has been dismissed.

GKGameCenterViewController

This View Controller is the main game view controller.

Name of event	What it does and when
Finished	It is called when the player has stopped interacting with the view controller.

GKMatch

The `GKMatch` class deals with connections from players.

Name of event	What it does and when
<code>DataReceived</code>	It is raised when data is received from the player.
<code>Failed</code>	It is called when the match cannot connect to other players.
<code>StateChanged</code>	It is raised when the player connects or disconnects from a match.

GKMatchmakerViewController

The `MatchMaker View Controller` deals with matches between devices.

Name of event	What it does and when
<code>DidFailWithError</code>	It is raised when the View Controller suffers an unrecoverable error.
<code>DidFindMatch</code>	It is raised when a peer-to-peer match is found.
<code>DidFindPlayers</code>	It is called when a hosted match is found.
<code>ReceivedAcceptFromHostedPlayer</code>	It is raised when a player accepts an invite to a hosted match.
<code>WasCancelled</code>	It is raised when the user cancels the matchmaking request.

GKSession

The `GKSession` class can be both local or external to the device.

Name of event	What it does and when
<code>ConnectionFailed</code>	It is raised when an attempt to connect to another peer fails.
<code>ConnectionRequest</code>	It is raised when attempting to connect to another peer.
<code>Failed</code>	It is sent when a serious error occurred in the session.
<code>PeerChanged</code>	It is received when a peer changes state.
<code>ReceiveData</code>	It is raised when data is received from the peer.

MKMapView

The `MapView` class of **MapKit** deals with the creation and display of maps and the addition of pointers onto the view (such as the user or places of interest).

Name of event	What it does and when
<code>CalloutAccessoryControlTapped</code>	It is raised when the user taps one of the annotation view accessories such as buttons.
<code>ChangedDragState</code>	It is raised when the state of one of the annotation views has changed.
<code>DidAddAnnotationViews</code>	It is called when one or more annotation views are added to a map.
<code>DidAddOverlayViews</code>	It is called when one or more overlay views are added to a map.
<code>DidChangeUserTrackingModel</code>	It is raised when the user tracking mode changes.
<code>DidDeselectAnnotationView</code>	It is called when an annotation view has been deselected.
<code>DidFailToLocateUser</code>	It is raised when an attempt to find a user's position fails.
<code>DidSelectAnnotationView</code>	It is called when one of the annotations has been selected.
<code>DidStopLocatingUser</code>	It is called when the locate user service has been stopped.
<code>DidUpdateUserLocation</code>	It is raised when the location of the user has been updated.
<code>LoadingMapFailed</code>	It is called when the map loading fails (typically caused by no GPS connection).
<code>MapLoaded</code>	It is raised when the map has loaded.
<code>RegionChanged</code>	It is raised when the region has changed.
<code>RegionWillChange</code>	It is raised when the region is changing.
<code>WillStartLoadingMap</code>	It is raised when the map is about to start loading.
<code>WillStartLocatingUser</code>	It is raised when the user is about to be located.

MPMediaPickerController

These events are attached to the `MediaPlayerController` View Controller.

Name of event	What it does and when
<code>DidCancel</code>	It is called when the user clicks on the Cancel button.
<code>ItemsPicked</code>	It is called when the user has selected a number of media items.

MFMailComposeViewController and MFMessageComposeViewController

The `MFMailComposeViewController` and `MFMessageComposeViewController` both have this event. It is used for composing either an e-mail or a message.

Name of event	What it does and when
<code>Finished</code>	It is emitted when the composition has finished.

PKAddPassesViewController

The `passkit` view controller is used for the storage of passwords.

Name of event	What it does and when
<code>Finished</code>	It is raised after the add-passes view controller has completed.

QLPreviewController

The `QuickLook` preview controller allows for a quick look at a file.

Name of event	What it does and when
<code>DidDismiss</code>	It is called after the preview controller is closed.
<code>WillDismiss</code>	It is called prior to the preview controller being closed.

SK classes

The `StoreKit` classes deal with the app store and online purchases from the app store.

SKProductsRequest

This class deals with requesting a product from the app store.

Name of event	What it does and when
ReceivedResponse	It is called when the App Store responds to the product request.
RequestFailed	It is raised when the request to the App Store fails.
RequestFinished	It is raised when the product request is closed.

SKRequest

This class is used for dealing with requests to the store.

Name of event	What it does and when
RequestFailed	It is raised when the request to the App Store fails.
RequestFinished	It is raised when the product request is closed.

SKStoreProductViewController

The `StoreProductViewController` is the main View Controller used for the app store content.

Name of event	What it does and when
Finished	It is raised when the product request is closed.

UIClasses

These classes deal exclusively with user interface events, and without them very little can be done.

UIAccelerometer

The accelerometer detects the movement of the device.

Name of event	What it does and when
Acceleration	It is raised when a movement is detected.

UIAlertSheet and UIAlertView

Both of these classes share these named events with the same effect in both.

Name of event	What it does and when
Cancelled	It is raised when the Cancel button is clicked on.
Clicked	It is raised when a control is selected.
Dismissed	It is raised when <code>ActionSheet</code> or <code>UIAlertView</code> is closed.
Presented	It is raised when <code>ActionSheet</code> or <code>UIAlertView</code> is shown.
WillDismiss	It is raised when <code>ActionSheet</code> or <code>UIAlertView</code> is about to be closed.
WillPresent	It is raised when <code>ActionSheet</code> or <code>UIAlertView</code> is about to be shown.

UIButtonBarItem

The **Button Bar Item** has to be connected to a `BarButton` to work.

Name of event	What it does and when
Clicked	It is called when the item is clicked.

UIImagePickerController

This class is used for picking images from the camera roll.

Name of event	What it does and when
Cancelled	It is raised when the picker has been cancelled.
DidShowViewController	It is raised when the View Controller has been shown.
FinishedPickingImage	It is raised when the user has finished picking image(s).
FinishedPickingMedia	It is raised when the user has finished picking media.
WillShowViewController	It is raised prior to the View Controller being displayed.

UIPageViewController

This class is used as a form of the "virtual" page counter.

Name of event	What it does and when
DidFinishAnimating	It is raised after the page scroll transition has completed.
WillTransition	It is raised before the page scroll transition has started.

UIPopoverController

This class is only available on the iPad.

Name of event	What it does and when
DidDismiss	It is raised when the controller has been dismissed.

UIPrintInteractionController

Used for printing from the device.

Name of event	What it does and when
DidDismiss	It is raised when the controller has been dismissed.
DidFinishJob	It is raised when the printer has finished a job.
DidPresentPrinterOptions	It is raised when the printer options have been displayed.
WillDismissPrinterOptions	It is raised when the printer options have been closed.
WillPresentPrinterOptions	It is raised when the printer options are about to be displayed.
WillStartJob	It is raised when the print job is about to start.

UIScrollView

The scroll view allows more content on a page that would not fit on the page without it.

Name of event	What it does and when
DecelerationEnded	It is called when the scroll deceleration has stopped.
DecelerationStarted	It is called when the scroll deceleration has started.
DidZoom	It is called once the zoom has occurred.
DraggingEnded	It is called when a drag has stopped (finger removed from phone).
DraggingStarted	It is called when a drag has started (finger dragging on phone).
ScrollAnimationEnded	It is called after a scroll has completed.
Scrolled	It is called when a scroll has completed.
ScrolledToTop	It is raised when a scroll to the top of a view has been completed.
WillEndDragging	It is raised when the user is about to end a drag.
ZoomingEnded	It is raised when the zoom has completed.
ZoomingStarted	It is raised when the zoom has started.

UISearchBar

This is a search method for finding information either within an app or on the device.

Name of event	What it does and when
BookmarkButtonClicked	It is called when the Bookmark button is clicked on.
CancelButtonClicked	It is called when the Cancel button is clicked on.
ListButtonClicked	It is called when the List button is clicked on.
OnEditingStarted	It is called when the UITextField view bounds are entered.
OnEditingStopped	It is called when the UITextField view bounds are left.
SearchButtonClicked	It is called when the Search button is clicked on.
SelectedScope ButtonIndexChanged	It is called when the scope button selection has changed.
TextChanged	It is called when the UITextField text has been changed.

UISplitViewController

The split view allows for a view to be split into parts (for example, a menu on the left appears when a menu button is pressed on a button bar).

Name of event	What it does and when
WillHideViewController	It is raised prior to the View Controller being hidden.
WillPresentViewController	It is raised prior to the View Controller being presented.
WillShowViewController	It is raised prior to the View Controller being shown.

UITabBar

A simple method of navigation using tabs can be used in association with the TabBarController controller.

Name of event	What it does and when
DidBeginCustomizingItems	It is raised after the customizing modal view is displayed.
DidEndCustomizingItems	It is raised after the customizing modal view is dismissed.
ItemSelected	It is called when a tab bar item is selected.
WillBeginCustomizingItems	It is raised before the customizing modal view is dismissed.
WillEndCustomizingItems	It is raised before the customizing modal view is dismissed.

UITabBarController

This is a convenient method of controlling the NIBs called when a tab bar item is clicked.

Name of event	What it does and when
FinishedCustomizingViewController	It is raised when the tab bar customization sheet is dismissed.
OnCustomizingViewController	It is raised when the customization has begun.
OnEndCustomizingViewController	It is raised when customization has ended.
ViewControllerSelected	It is called when the user selects an item on the tab bar.

UITextField

The `UITextField` class is a simple, editable textbox.

Name of event	What it does and when
Ended	It is raised when editing of the <code>TextField</code> has ended.
Started	It is raised as soon as the content of the <code>TextField</code> is edited.

UITextView

The `UITextView` class displays text. It also inherits from the `ScrollView` view to enable more text than within the frame.

Name of event	What it does and when
Changed	It is raised when the text has changed.
DecelerationEnded	It is called when the scroll deceleration has stopped.
DecelerationStarted	It is called when the scroll deceleration has started.
DidZoom	It is called once the zoom has occurred.
DraggingEnded	It is called when a drag has stopped (finger removed from phone).
DraggingStarted	It is called when a drag has started (finger dragging on phone).
Ended	It is called after a scroll has been completed.
ScrollAnimationEnded	It is called when a scroll has been completed.
Scrolled	It is raised when the text view is scrolled.

Name of event	What it does and when
ScrolledToTop	It is raised when a scroll to the top of a view has been completed.
SelectionChanged	It is raised when the selection of text within the text view has changed.
Started	It is raised when a scroll has started.
WillEndDragging	It is raised when a user is about to end a drag.
ZoomingEnded	It is raised when a zoom has completed.
ZoomingStarted	It is raised when a zoom has started.

UIView

The `UIView` is a generic view that can be added to any View Controller.

Name of event	What it does and when
AnimationWillEnd	It is raised when an animation is about to end.
AnimationWillStart	It is raised when an animation is about to start.

UIWebView

A view for displaying HTML (either from a website or generated within an app).

Name of event	What it does and when
LoadError	It is raised when URL loading has failed.
LoadFinished	It is raised when URL has finished loading.
LoadStarted	It is raised when URL loading has begun.

Ad classes

Typically, advertisements within applications are a convenient method of generating income for the developer (it's known as a click-through).

AdBannerView

The AdBannerView View is the view containing ads.

Name of event	What it does and when
ActionFinished	It is raised when the banner view finishes its execution of an action that covered the UI.
AdLoaded	It is raised when an ad is loaded.
FailedToReceiveAd	It is raised when an ad retrieval has failed (typically a network connection error).
WillLoad	It is raised when an ad is about to load.

AdInterstitialAd

These are full-screen advertisements. They act in the same way as AdBannerView.

Name of event	What it does and when
ActionFinished	It is raised when the banner view finishes its execution of an action that covered the UI.
AdLoaded	It is raised when an ad is loaded.
AdUnloaded	It is called after a full-screen ad disposes its content.
FailedToReceiveAd	It is raised when the ad retrieval has failed (typically a network connection error).
WillLoad	It is raised when the ad is about to load.

OpenTK

OpenTK is an open graphics layer used commonly among mobile and desktop application developers.

IGameWindow

This is an interface class rather than a class and deals with the game window itself rather than the game.

Name of event	What it does and when
Load	It is raised before a window is displayed for the first time.
RenderedFrame	It is raised when the time to render has arrived.
Unload	It is raised when a window is destroyed.
UpdateFrame	It is called when it's time to update a frame.

iPhoneOSGameView

This is the `GameView` held by the `OpenTK` classes.

Name of event	What it does and when
<code>Closed</code>	It is called when the game view has been closed but not disposed
<code>Disposed</code>	It is called when the game view has been disposed
<code>Load</code>	It is raised before the run loop starts.
<code>RenderedFrame</code>	It is raised as part of the run-loop processing for when a frame should be rendered.
<code>Resize</code>	It is called when a view is resized.
<code>TitleChanged</code>	It is called when a view title is changed.
<code>Unload</code>	It is raised when a run-loop is terminated.
<code>UpdateFrame</code>	This is raised when a frame is updated as part of the runloop.
<code>VisibleChanged</code>	It is called when the visibility of a view is changed.
<code>WindowStateChanged</code>	It is raised when a window state changes.

Summary

It is safe to say that events are what make your iPhone the device it is. It is very unlikely that you'll ever need most of these listed here, but if you're like me and hate having to search, this chapter should really help you in the future.

7

Gestures

While there are ongoing arguments in the courts of America at the time of writing over who invented the likes of dragging images, it is without a doubt that a key feature of iOS is the ability to use **gestures**. To put it simply, when you tap the screen to start an app or select a part of an image to enlarge it or anything like that, you are using gestures.

We will be covering the following topics in this chapter:

- What is a gesture?
- Adding gestures to the UI
- Handling gestures
- Handling drag-and-drop

Gestures

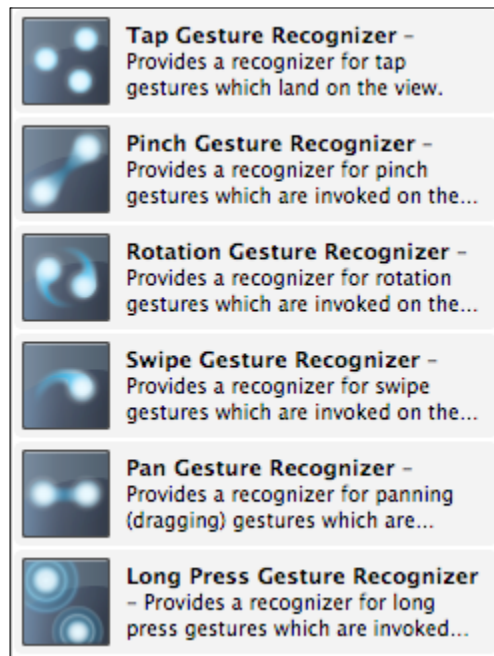
A gesture (in terms of iOS) is any touch interaction between the UI and the device. With iOS 6, there are six gestures the user has the ability to use. These gestures, along with brief explanations, have been listed in the following table:

Class	Name and type	Gesture
UIPanGestureRecognizer	PanGesture; Continuous type	Pan images or over-sized views by dragging across the screen
UISwipeGestureRecognizer	SwipeGesture; Continuous type	Similar to panning, except it is a swipe
UITapGestureRecognizer	TapGesture; Discrete type	Tap the screen a number of times (configurable)

Gestures

Class	Name and type	Gesture
UILongPressGestureRecognizer	LongPress Gesture; Discrete type	Hold the finger down on the screen
UIPinchGestureRecognizer	PinchGesture; Continuous type	Zoom by pinching an area and moving your fingers in or out
UIRotationGestureRecognizer	RotationGesture; Continuous type	Rotate by moving your fingers in opposite directions

Gestures can be added by programming or via Xcode. The available gestures are listed in the following screenshot with the rest of the widgets on the right-hand side of the designer:



To add a gesture, drag the gesture you want to use under the view on the View bar (shown in the following screenshot):



Design the UI as you want and while pressing the *Ctrl* key, drag the gesture to what you want to recognize using the gesture. In my example, the object you want to recognize is anywhere on the screen. Once you have connected the gesture to what you want to recognize, you will see the configurable options of the gesture.

The **Taps** field is the number of taps required before the Recognizer is triggered, and the **Touches** field is the number of points onscreen required to be touched for the Recognizer to be triggered.

When you come to connect up the UI, the gesture must also be added.

Gesture code

When using Xcode, it is simple to code gestures. The class defined in the Xcode design for the tapping gesture is called `tapGesture` and is used in the following code:

```
private int tapped = 0;
public override void ViewDidLoad()
{
    base.ViewDidLoad();
    tapGesture.AddTarget(this, new Selector("screenTapped"));
    View.AddGestureRecognizer(tapGesture);
}
```

```
[Export("screenTapped")]
public void SingleTap(UIGestureRecognizer s)
{
    tapped++;
    lblCounter.Text = tapped.ToString();
}
```

There is nothing really amazing to the code; it just displays how many times the screen has been tapped.

The `Selector` method is called by the code when the tap has been seen. The method name doesn't make any difference as long as the `Selector` and `Export` names are the same.

Types

When the gesture types were originally described, they were given a type. The type reflects the number of messages sent to the `Selector` method. A discrete one generates a single message. A continuous one generates multiple messages, which requires the `Selector` method to be more complex. The complexity is added by the `Selector` method having to check the **State** of the gesture to decide on what to do with what message and whether it has been completed.

Adding a gesture in code

It is not a requirement that Xcode be used to add a gesture. To perform the same task in the following code as my preceding code did in Xcode is easy. The code will be as follows:

```
UITapGestureRecognizer t'pGesture = new UITapGestureRecognizer()
{
    NumberOfTapsRequired = 1
};
```

The rest of the code from `AddTarget` can then be used.

Continuous types

The following code, a Pinch Recognizer, shows a simple rescaling. There are a couple of other states that I'll explain after the code. The only difference in the designer code is that I have `UIImageView` instead of a label and a `UIPinchGestureRecognizer` class instead of a `UITapGestureRecognizer` class.

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();
    UIImageView.Image = UIImage.FromFile("graphics/image.jpg")
        Scale(new.SizeF(160f, 160f));
    pinchGesture.AddTarget(this, new Selector("screenTapped"));
    UIImageView.AddGestureRecognizer(pinchGesture);
}
[Export("screenTapped")]
public void SingleTap(UIGestureRecognizer s)
{
    UIPinchGestureRecognizer pinch = (UIPinchGestureRecognizer)s;
    float scale = 0f;
    PointF location;
    switch(s.State)
    {
        case UIGestureRecognizerState.Began:
            Console.WriteLine("Pinch begun");
            location = s.LocationInView(s.View);
            break;
        case UIGestureRecognizerState.Changed:
            Console.WriteLine("Pinch value changed");
            scale = pinch.Scale;
            UIImageView.Image = UIImage
                FromFile("graphics/image.jpg")
                Scale(new.SizeF(160f, 160f), scale);
            break;
        case UIGestureRecognizerState.Cancelled:
            Console.WriteLine("Pinch cancelled");
            UIImageView.Image = UIImage
                FromFile("graphics/image.jpg")
                Scale(new.SizeF(160f, 160f));
            scale = 0f;
            break;
        case UIGestureRecognizerState.Recognized:
            Console.WriteLine("Pinch recognized");
            break;
    }
}
```

Other UIGestureRecognizerState values

The following table gives a list of other Recognizer states:

State	Description	Notes
Possible	Default state; gesture hasn't been recognized	Used by all gestures
Failed	Gesture failed	No messages sent for this state
Translation	Direction of pan	Used in the pan gesture
Velocity	Speed of pan	Used in the pan gesture

In addition to these, it should be noted that discrete types only use Possible and Recognized states.

Handling drag-and-drop

Drag-and-drop can be handled using a gesture or by using the `TouchesBegan`, `TouchesMoved`, and `TouchesEnded` handlers. Essentially, a custom `UIImageView` class can be used, as shown in the following code:

```
public class myDragImage : UIImageView
{
    private PointF myLoc, myStartLoc;
    private bool TouchedOnce = false;

    public myDragImage (RectangleF frame)
    {
        this.Frame = frame;
        myStartLoc = this.Frame.Location;
    }
    public override void TouchesBegan(NSSet touches, UIEvent e)
    {
        myLoc = Frame.Location;

        var touch = (UITouch)e.TouchesForView(this).AnyObject;
        var bounds = Bounds;

        myStartLoc = touch.LocationInView(this);
        Frame = new RectangleF(Location,bounds.Size);
    }
}
```

```
public override void TouchesMoved(NSSet touches, UIEvent e)
{
    var bounds = Bounds;
    var touch = (UITouch)e.TouchesForView(this).AnyObject;

    myLoc.X += touch.LocationInView(this).X - myStartLoc.X;
    myLoc.Y += touch.LocationInView(this).Y - myStartLoc.Y;

    Frame = new RectangleF(myLoc, bounds.Size);
    TouchedOnce = true;
}

public override void TouchesEnded(NSSet touches, UIEvent e)
{
    myStartLoc = myLoc;
}
}
```

This is used as a simple way to handle drag-and-drop. For a gesture, a Continuous type should be used.

Summary

Gestures certainly can add a lot to your apps. They can enable the user to speed around an image, move about a map, enlarge and reduce, as well as select areas of anything on a view. Their flexibility underpins why iOS is recognized as being an extremely versatile device for users to manipulate images, video, and anything else on-screen.

8

Threading

iOS is what is known as a multithreading system, and understanding how threads can be used within an app can be advantageous.

We will be covering the following topics in this chapter:

- A brief introduction to threading
- The main UI thread
- A Daughter thread
- The AppDelegate class

Threading Concepts

Let's discuss an easy way to learn threading.

A single-thread environment can be considered in the same way as going to your local college. There are a number of routes you can take, but you end up there at some point and the process will take a finite amount of time; you set off, you travel, you arrive.

A multithreaded environment needs to be thought of as the college itself with each thread being a student. All students start off at 9 a.m. and go until 12 p.m. What they do in between that time may or may not interfere with each other; they will all be doing a task or co-operating on a task to speed up the delivery of an answer. Thirty different threads, all working at once and at different speeds, but at 12 p.m., they all manage to converge and terminate their activities with the jobs done. They repeat the process from 1 p.m. to 4 p.m., and again, there is organized chaos between those hours, but at 4 p.m. everything converges. The lecturer is the one controlling who does what, and in terms of the threading model, is the control thread. *Simple!*

In terms of iOS development, the lecturer would be classed as the UI thread; it is the one that can start new threads and, at the end of the day, the one where all information needs to be fed back.

The main UI thread

As the name suggests, the UI thread controls the UI. It is usually the hungriest in terms of resources and processor time. Not everything runs on the UI thread. For example, if the UI calls a method and that method cannot be run on the UI thread (such as the SQLite example listed in *Chapter 11, Handling Data*), then that is what will happen. The code is executed and the flow continues once the method has returned.

The UI thread should not be mistaken as a single task; *it's not*. A single task would prohibit any other application running, which we know is not the case (for example, you could be playing Angry Worms and still receive a text message).

Xamarin.iOS allows non-UI calls to simply jump back onto the UI thread.

```
InvokeOnMainThread(delegate () {...});
```

Or, if a reference to the thread can't be found (such as being out of scope or in a non-UI thread class), use the following line of code:

```
using (var pool = new NSAutoreleasePool()) {
    pool.InvokeOnMainThread(delegate() {
        // do something on the UI thread
    });
}
```

Deadlocking

Something may have crossed your mind over the description of the multithreaded system. What happens if all the students don't come back when they should? What happens then? It's a good question as it's something that if you're not careful can hit when dealing with a multithreaded environment. It is known as a deadlock and, literally, it can lock the app and potentially the device (though this is rare). Another problem is threads overwriting the same memory location (think of this as two or more people talking to the lecturer at once; only one voice will be remembered).

In this example, if the two threads are run within a second of each other, they will both have time to grab the first lock before anyone gets to the inner lock. Without the `Sleep()` call, one of the threads would most likely have time to get and release both locks before the other thread even got started.

```
// thread 1
lock(typeof(int)) {
    Thread.Sleep(1000);
    lock(typeof(float)) {
        Console.WriteLine("Thread 1 got both locks");
    }
}

// thread 2
lock(typeof(float)) {
    Thread.Sleep(1000);
    lock(typeof(int)) {
        Console.WriteLine("Thread 2 got both locks");
    }
}
```

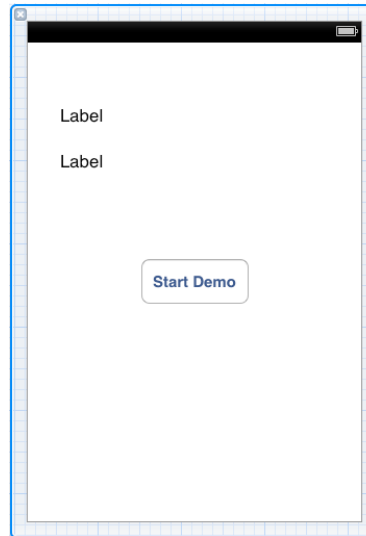
Avoiding deadlocks for synchronized accessors

A simple way to avoid this form of deadlock is for each holding class to have its own private deadlock. The problem along with its solution is described well on MSDN (<http://msdn.microsoft.com/en-us/library/orm-9780596516109-03-18.aspx>).

Starting a new thread from the main UI thread

A new thread coming from an existing thread is known as a daughter thread.

A very simple way to add a daughter thread on an iOS device is like this. I have first created a simple UI to show what is happening. The top label is called thread 1, the bottom label is called thread 2.



The code also shows `InvokeOnMainThread` in action—without it the app fails:

```
using System.Threading;
...
private int i = 0;

public override void ViewDidLoad() {
    base.ViewDidLoad();
    var first = new Thread(new ThreadStart(firstThread));
    var second = new Thread(new ThreadStart(secondThread));
    btnStart.TouchUpInside += delegate {
        first.Start();
        Thread.Sleep(10);
        // causes a 10ms delay between starting the next thread

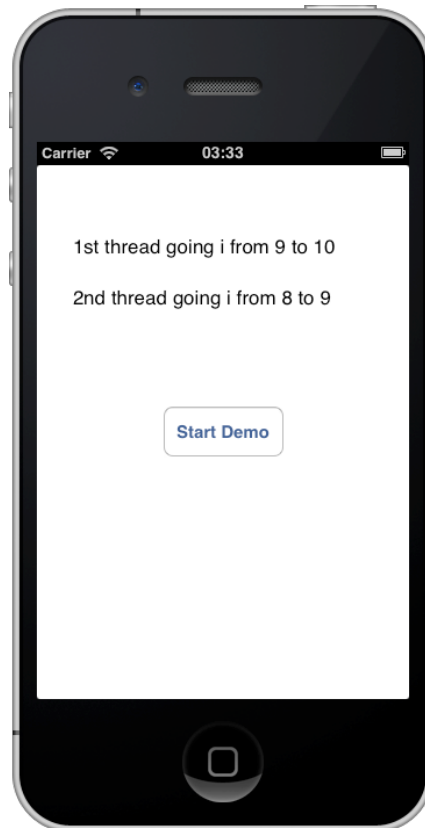
        second.Start();
    };
}

private void firstThread() {
    string text = string.Empty;
    while (i < 10) {
        text = string.Format("1st thread going i from {0} to {1}",
            i, ++i);
    }
}
```

```
        InvokeOnMainThread(delegate() {
            thread1.Text = text;
        });
        Thread.Sleep(100);
    }
}

private void secondThread() {
    string text = string.Empty;
    while (i < 10) {
        text = string.Format("2nd thread going i from {0} to {1}",
            i, ++i);
        InvokeOnMainThread(delegate() {
            thread2.Text = text;
        });
        Thread.Sleep(100);
    }
}
```

And when run, the simulator gives the following output:



Run this a number of times and you get a number of different results. The threads are performing operations on the UI at different times; this shows the problem with threading quite well. If the UI was waiting for thread 1 to finish but thread 2 finishes, then it's not going to know what is going on.

In this case, the code can be sanitized by using a lock.

Using locks

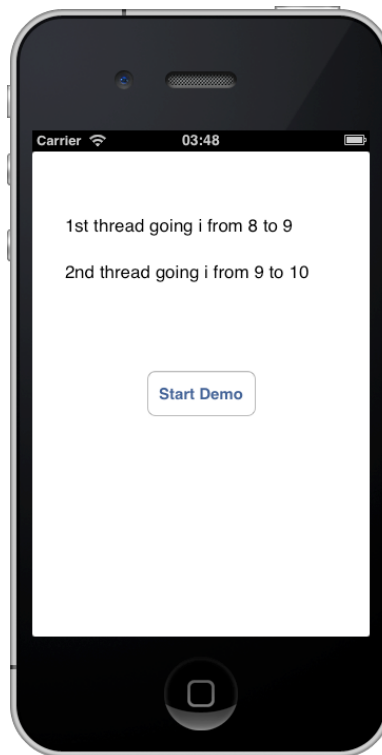
Be careful when using locks is probably the first thing that needs to be said. A lock is used to synchronize the threads and obtain a far saner output.

```
private int i = 0;
private object lock_i = new object();
public override void ViewDidLoad() {
    base.ViewDidLoad();
    var first = new Thread(new ThreadStart(firstThread));
    var second = new Thread(new ThreadStart(secondThread));
    btnStart.TouchUpInside += delegate {
        first.Start();
        Thread.Sleep(10);
        // causes a 10ms delay between starting the next thread

        second.Start();
    };
}
private void firstThread() {
    string text = string.Empty;
    do {
        lock(this.lock_i) {
            if (i >= 10) return;
            text = string.Format("1st thread going i from {0} to {1}",
                i, ++i);
            InvokeOnMainThread(delegate() {
                thread1.Text = text;
            });
        }
        Thread.Sleep(100);
    }
    while(true);
}
```

```
private void secondThread() {
    string text = string.Empty;
    do {
        lock(this.lock_i) {
            if (i >= 10) return;
            text = string.Format("2nd thread going i from {0} to {1}",
                i, ++i);
            InvokeOnMainThread(delegate() {
                thread2.Text = text;
            });
        }
        Thread.Sleep(100);
    }
    while(true);
}
```

This time when the app is run, the threads are synchronized and the result is always the same. Using this locking system, the app is free to use as many threads as it needs to get whatever done off the UI thread.



The AppDelegate class

It may seem odd having the AppDelegate class described here, but it fits. The AppDelegate class is known as a singleton class. It's used once and once only with everything coming from it. Consider it as the über thread; without it, nothing else happens.

I've given the AppDelegate class a more thorough handling in *Chapter 5, UI Controls* and, after reading this chapter, you should have a clearer idea of its importance.

Summary

Threading within an iOS application can make up for a more responsive user experience, but at the same time, for the developer it can be the reason for many a late nights trying to figure out why something is crashing at random times or just seizes up for no real reason. Be careful with threads, they can be both a pain and a pleasure.

9

Threading Tasks

In the previous chapter, we took a look at the basics of using threads within an iOS application and the pitfalls that may confront you if you use them. In this chapter, I'll be carrying it on and we'll have a look at the other aspects of threading as well as asynchronous calls.

In this chapter we will be covering the following topics:

- Using background threading and `System.Threading.Tasks` within your code
- Using asynchronous code
- Problems that using tasks may have on the threading model

A brief introduction to threading

Threading moved on from its humble beginnings when developers discovered its power, and with that created background threading and task threading. Background threads are just that – you set something running in the background and look in on it sometimes, or when it's finished it will report back to you. On the college analogy I used in the previous chapter, the background threads are the admin staff – they're there in the background working away and report when they are done.

Threading tasks need to be thought of as almost miniature applications in themselves. They start, end, and can continue with the next task on the list – all this time, the app is free to be working on other tasks. There is an overhead to be considered with any threading operation, but unless you're doing something insanely complex, it's not going to be horrible.

Using background threading within your app

Background threading comes from the `System.ComponentModel` namespace and is known as a `BackgroundWorker` thread. Alternatively, `ThreadPool.QueueUserWorkItem()` does the same thing (as `ThreadPool` is from `System.Threading`).

BackgroundWorker

The `BackgroundWorker` thread is recommended when you don't want to tie up the UI, so creating large files or sending a large amount of data to a server can be considered to be used with `BackgroundWorker`. When the thread is complete, the `WorkerCompleted` event is raised. During the operation of `BackgroundWorker`, the UI can be updated with the `ProgressChanged` event. A background lasts for a finite amount of time. It is important to remember that `BackgroundWorker` is an asynchronous task.

When you use a `BackgroundWorker` thread, you need to write a code using three events (`ProgressChanged` can be omitted if you don't want to use it).

```
DoWork(object sender, DoWorkEventArgs e);
RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e);
ProgressChanged(object sender, ProgressChangedEventArgs e);
```

The preceding code demonstrates using `BackgroundWorker` within an app. It's simple enough; it puts a counter on the screen, which will carry on counting while it downloads a picture that is then displayed when the `RunWorkerCompleted` event is raised.

```
private UIImage downloadedImage;
private BackgroundWorker bgWorker;
private Timer t;
private int counter = 0;

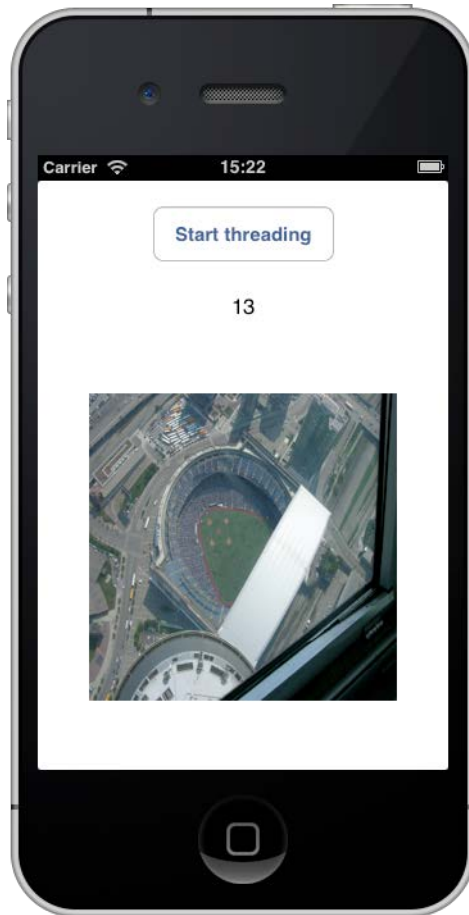
public override void ViewDidLoad()
```

```
{
    base.ViewDidLoad();
    bgWorker = new BackgroundWorker();
    bgWorker.DoWork += HandleDoWork;
    bgWorker.RunWorkerCompleted += HandleRunWorkerCompleted;
    btnStart.TouchUpInside += delegate
    {
        t = new Timer(1000); // 1 second
        t.Elapsed += delegate
        {
            counter++;
            InvokeOnMainThread(delegate()
            {
                lblCountValue.Text = counter.ToString();
            });
        };
        t.Start();
        bgWorker.RunWorkerAsync();
    };
}

private void HandleRunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
{
    UIImageView.Image =
        UIImage.FromImage(downloadedImage.CGImage).Scale(new
        SizeF(240f, 240f));
    t.Stop();
}

private void HandleDoWork(object sender, DoWorkEventArgs e)
{
    NSURL url = new NSURL("http://edmullen.net/test/rc.jpg");
    NSData data = NSData.FromUrl(url);
    downloadedImage = new UIImage(data);
}
```

The preceding code is simple enough to follow, it creates the `BackgroundWorker` object and it also creates the handles and a click event for the button. Inside the button, it sets up a timer to update the counter every second and start `BackgroundWorker`. When `BackgroundWorker DoWork` thread is completed, the image is displayed and scaled. The result of the preceding code is shown in the form of the following image:



Remember though, this is a background task – the UI can't act on the data until the data is ready. When you run the application, the counter value will differ as well, depending on if you're on a wireless network or outside, using 3G or 4G.

ThreadPool.QueueUserWorkItem

Having seen how the background worker operates, let us consider using `ThreadPool.QueueUserWorkItem` for the same task:

```
private UIImage downloadedImage;
private System.Timers.Timer t;
private int counter = 0;

public override void ViewDidLoad()
{
    base.ViewDidLoad();
    btnStart.TouchUpInside += delegate
    {
        t = new Timer(100);
        t.Elapsed += delegate
        {
            counter++;
            InvokeOnMainThread(delegate()
            {
                lblCountValue.Text = counter.ToString();
            });
        };
        t.Start();
        ThreadPool.QueueUserWorkItem(delegate
        {
            ProcessFile();
        });
    };
}

private void ProcessFile()
{
    NSUrl url = new NSUrl("http://edmullen.net/test/rc.jpg");
    NSData data = NSData.FromUrl(url);
    downloadedImage = new UIImage(data);
    InvokeOnMainThread(delegate()
    {
        UIImageView.Image =
            UIImage.FromImage(downloadedImage.CGImage).Scale(new
               .SizeF(240f, 240f));
    });
    t.Stop();
}
```

The **callback** is a piece of code passed as an argument, which needs to be executed at some point in time. In threading terms, it is typically executed within the thread being called or created. The callback tells the thread when it has called the thread back to the main thread. The end result is the same, but the `QueueUserWorkItem` method can be used for both foreground and background tasks.

Using System.Threading.Tasks

The `System.Threading.Tasks` namespace sets up tasks within a thread, so a thread can perform a type of mini program and then report back. It can also be used to start a task.

```
var scheduler = TaskScheduler.FromCurrentSynchronizationContext();
Task.Factory.StartNew(() => GetMessage(currentPosition))
    .ContinueWith(ShowResults, scheduler);
```

The call starts a new thread task that calls `GetMessage`. Once that has returned, the task continues with `ShowResults`. The scheduler prevents the timing from getting out of hand.



While the code available at <http://www.gregshackles.com/2011/04/using-background-threads-in-mono-for-android-applications/> is for Android, the same code can (more or less) be used in Xamarin.iOS and gives a fantastic cover of the different types of threading and how they're used.

Problems while using Tasks on threads

Whenever an additional task is created, the processor has to start swapping between the tasks themselves, which slows the code down. You have the additional problem of tracking the tasks and how they work with the main UI thread. In general, they shouldn't cause a problem, but you also have to consider that unless you explicitly program the threads to run synchronously, they run asynchronously. To prevent the code from getting out of hand, locks or callbacks need to be used. **Locks** may lead to **deadlock** conditions, so be careful! Refer back to *Chapter 8, Threading*, for an overview of deadlocks and avoiding them.

Using Asynchronous code

Async is one of the big changes to .NET and was released in .NET v4, but has only quite recently landed within the Mono framework, and therefore within Xamarin. Android and Xamarin.iOS. As I explained in a previous chapter, asynchronous code can be a bit of a handful, but thankfully that bit of a handful is simple to understand.

Tasks and EventHandlers

Take the following code as an example:

```
var webView = new UIWebView();
webView.LoadStarted += HandleLoadStarted;
...
private void HandleLoadStarted (object sender, EventArgs e)
{ }
```

The handler for `LoadStarted` is a synchronous process – in other words, it is like a walk to the pub in a straight line. The problem is that while `webView` is loading a page, everything is being held up – so if it's a slow page or you need something else to be running (say a piece of music to play), there are going to be sticking points. This is where an asynchronous `LoadStarted` event can be used.

```
var webView = new UIWebView();
webView.LoadStarted += async(object sender, EventArgs e) =>
    {HandleLoadStarted(sender, e);};
```

The `async` method looks different from a normal method as shown in the following code:

```
private async void HandleLoadStarted(object sender, EventArgs e)
```

A more practical example

Prior to the `async` method being implemented, a system of events had to be implemented so that once the data had been returned it can be handled by the method (say, from a web data download). For example (the following is a psuedo code, so you get the idea):

```
login.name = "nodoid201213";
login.password = "312102diodon";
login.DataReturned += HandleDataReturned;
callLoginService(login);
private void HandleDataReturned { ... }
```

The preceding code can now be handled in a single `async` method, as shown in the following code:

```
private async Task<bool> LogUserIn()
{
    login.name = "nodoid201213";
    login.password = "312102diodon";
    bool loginResult = await callLoginService(login);
    return loginResult;
}
```

The key here is `await` — this prevents the next line from executing until `callLoginService` has returned. This greatly improves responsiveness — less code, fewer events to listen to, and far less messing about.

If a method returns a value, the `Task<T>` parameter needs to be used prior to the method name. If there isn't a return value (as, when responding to a button click), `void` needs to be used.

Summary

There are many uses for background threads as well as asynchronous calls, and the general recommendation is that if a process takes a long time, throw it at the background. Be careful when using threads. While for the majority of the time they are fine, you still need to test all apps on a real device to ensure the threads are working. Remember, the simulator is buggy (for example, the simulator works on a **Just In Time** processor model rather than **Ahead Of Time**, which the phone uses — the results are that web services may not work as planned) and doesn't work the same way as a phone.

10

Animation

Animation is the illusion of movement of static (still) images. To do this, typically something has to move roughly every 1/25th of a second. The more the steps used for moving something, the smoother the motion and the easier it is to fool the brain. We've already seen *Chapter 6, Events*, how animation can be achieved using `UIAlertView`. Now we need to see how we can do this normally using the `CoreAnimation` and `CoreGraphics` namespaces. This is not going to be an exhaustive study but it will give you a grounding in the basics.

In this chapter, we will be covering the following:

- Handling bitmaps (scaling and rotation)
- Freeing memory after use

Handling bitmaps

A bitmap image can be created either inside or outside of an app. External bitmaps are rendered to `UIImageView` as shown in the following code:

```
imageView = UIImage.FromFile("path/tofile.ext");
```

We can get the image in, so let's do something with it.

Scaling the image

Scaling can be achieved by setting the scale factor.

```
imageView = UIImage.FromFile("path/tofile.ext").Scale(  
    new.SizeF(float w, float h), float scaleFactor);
```

If `scaleFactor` is not specified, it is 1.0f by default.

Therefore, you could create a kind of animation as follows:

```
float w = 10f, h = 10f;
for (int i = 0; i < 100; ++i) {
    UIImageView = UIImage.FromFile("path/tofile.ext").Scale(
        new.SizeF(w, h));
    w += (float)i + 5f;
    h += (float)i + 5f;
}
```

This gives the impression of the image growing. It's not very good, but gives you an idea. To get further than this (such as rotation), we need to start looking at `CoreGraphics` and `CoreAnimation`.

Rotating the image – Part 1

I will assume here that there is a small-enough `UIImageView` widget set onto a view (say 122 x 122) in the middle of the screen. As before, the image is loaded in, but this time the `CoreGraphics` image is required.

```
UIImageView.Image = UIImage.FromFile("graphics/image.jpg").Scale(
    new.SizeF(122f, 122f));
UIImageView.Transform = CGAffineTransform.MakeRotation(
    (float)Math.PI / 15f);
```

The image is loaded and rotated. The rotation is static (in other words, instant).

For animation, `CoreAnimation` needs to be used. However, before doing a rotation animation, let's start off on something simpler – moving something across the screen and back. To do this, let's look at some code.

```
private PointF startPoint;

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

    UIImageView.Image = UIImage.FromFile(
        "graphics/image.jpg").Scale(new.SizeF(122f, 122f));
    startPoint = UIImageView.Center;

    UIView.BeginAnimations("moveImage");
    UIView.SetAnimationDuration(2);
    UIView.SetAnimationCurve(UITableViewAnimationCurve.EaseInOut);
    UIView.SetAnimationRepeatCount(2);
```

```
UIView.SetAnimationRepeatAutoreverses(true);
UIView.SetAnimationDelegate(this);
UIView.SetAnimationDidStopSelector(new Selector(
    "moveImageStopped:"));
UIImageView.Center = new PointF(
    UIScreen.MainScreen.Bounds.Right - UIImageView.Frame.Width /
    2, UIImageView.Center.Y);
UIView.CommitAnimations();
}

[Export("moveImageStopped")]
private void moveImageStopped() {
    UIImageView.Center = startPoint;
}
```

There are two important points to note about this code.

- The code operates on `UIView` rather than `UIImageView`
- The bindings between Xamarin.iOS and the underpinning Objective-C become very visible for animation and drawing in general (the binding is the selector)

Underpinning bindings

In the preceding example, the code is creating an interface layer for the underpinning Objective-C. The compiler handles this in a slightly different manner compared to normal code. Adding this sort of `Selector` code can be used in other ways as well (for example, to access private API code – though this should be avoided as it will debar apps from being accepted into the app store). It should be noted that the bindings to the Objective-C layer may sometimes cause issues with submission to the Apple store.

Analysis of the code

The analysis of the preceding code can be summed up as follows:

- `startPoint` is the position of the image at the start.
- To tell the app there is going to be an animation, `BeginAnimations` needs to be called.
- `Duration` is the length of the animation and `RepeatCount` is the number of times the animation is called.

- `RepeatAnimationCurve` defines how the animation is to proceed (in this case, to repeat the animation curve, a curve does not have to be an arc on a circle, it can be a straight line).
- `EaseInOut` starts the animation slowly and builds up and slows down.
- `EaseIn` starts the animation slowly.
- `EaseOut` slows it at the end
- `Linear` gives a uniform speed.
- The binding resets the image to the center once the animation has ended.
- `CommitAnimations` sets the animation going. Xamarin.iOS provides a very good example of animation using blocks that will provide further support on this topic.

Freeing memory after use

Typically, once a class has gone out of scope, the **garbage collector (GC)** will free up memory used by the processes within that class. However, as Xamarin.iOS works as a binding layer to the underpinning Objective-C, there are times when freeing memory becomes important; this mostly happens when dealing with animation and graphics.

Probably the simplest way to clean up is provided when a new View Controller is created.

```
public override void DidReceiveMemoryWarning() {  
    // Releases the view if it doesn't have a superview.  
    base.DidReceiveMemoryWarning();  
    // Release any cached data, images, etc that aren't in use.  
}
```

For example, to release `UIImageView` do as follows:

```
imageView.Release();
```

If the code doesn't cause a memory warning, `ViewDidDisappear()` can also be used to free the memory in the same way.

Another simple method of freeing memory is to allow the GC to do its job once the code has gone out of scope. Consider the following (simplistic) code:

```
private async void doSomething() {  
    UIImageView image = new UIImageView(new RectangleF(0, 0, 100,  
        100));  
}
```

```
string filename = await GetFileName();
image.Image = UIImage.FromFile(filename);
// do a lot of bits and pieces
if (condition)
    return;
else
    callNewMethod();
}
```

The code executes and loads `UIImageView` with the image as directed by the returned string. If `condition` is met (that is, it's `true`), the method returns. If `condition` is not met, the method jumps to `callNewMethod`. Neither of these are big issues, except that the GC does not get called until the class itself goes out of scope. So any memory occupied by the `UIImageView` control is still used, despite it only being used for three lines in one class. With too many images and too many manipulations, memory soon vanishes.

If you consider an average animation, there may be 300 images with backgrounds and so the memory is soon drained.

A simple solution is to only create and use what you need and use code that calls the GC once it has gone out of scope. The following lines of code demonstrate how to do this:

```
private async void doSomething() {
    using (UIImageView image = new UIImageView()) {
        image.Frame = new RectangleF(0, 0, 100, 100);
        string filename = await GetFileName();
        image.Image = UIImage.FromFile(filename);
    };
    // do a lot of bits and pieces
    if (condition)
        return;
    else
        callNewMethod();
}
```

While this looks similar, the image being created is used and once completed, the memory being used is freed up again, rather than having to wait until the class goes out of scope.

Rotating the image – Part 2

To get an image to rotate, the CoreGraphics image has to be used followed by conversion to a bitmap. The following gives you an idea of how to do the rotation. Altering `RotateCTM` and `TranslateCTM` from positive to negative (and vice versa) should give different results.

```
public static UIImage rotateImage(UIImage uiImage) {
    UIImage result;
    using (CGImage cgImage = uiImage.CGImage) {
        CGImageAlphaInfo alpha = cgImage.AlphaInfo;
        CGColorSpace colour = CGColorSpace.CreateDeviceRGB();
        if (alpha == CGImageAlphaInfo.None)
            alpha = CGImageAlphaInfo.NoneSkipLast;
        int width = cgImage.Width, height = cgImage.Height;
        CGContext bitmap = new CGContext(
            IntPtr.Zero, height,
            width, cgImage.BitsPerComponent, cgImage.BytesPerRow,
            colour, alpha);
        bitmap.RotateCTM((float)Math.PI / 2); // rotate right.
        bitmap.TranslateCTM(0, -height);
        bitmap.DrawImage(new Rectangle(0, 0, width, height), cgImage);
        result = UIImage.FromImage(bitmap.ToImage());
        bitmap = null; // free memory
    }
    return result;
}
```

Summary

Animation and graphics handling is an extensive topic on iOS. While this chapter has been a whistle-stop over the subject, I would recommend you have a look at *Learning MonoTouch* by Michael Bluestein, Pearson Education, Inc. His book covers the topic in much greater detail than space allows here.

11

Handling Data

From time to time it is important for an application to store or manipulate data. With the advent of LINQ, manipulation is extremely simple now. The problem, though, is that you need to store the data somehow. Thankfully this too is simple using SQLite.

We will be covering the following topics in this chapter:

- Using SQLite
- Setting up an SQLite helper class
- Using LINQ
- Dangers of using LINQ on iOS

Using SQLite

SQLite is a very simple database system that is also extremely powerful. It is outside the scope of this book to give you a master class on using SQLite, but understanding how to set up and use the system will help.

Installing and setting up SQLite

Installing can be performed in one of two ways; either you can install from the Xamarin component market (it is useful as it supplies you with examples and also the Android version) or you can download and install the software manually. As there are no additional libraries (SQLite comes as a single C# file) either way is good.

Once you have either copied the C# file or installed the component, the `SQLite.net` implementation is ready for use. It is as simple as inserting the following `using` directive at the top of the source file, and it's done:

```
using SQLite;
```

Database basics

An easy way to consider a database is like an old card index system (commonly known as a **cardex** system). Information (data) can be added, updated, read, or deleted – and SQLite gives you that facility within the mobile environment. The data is stored in a file with tables (the table can be considered as the box that holds the cards) holding the information you want.

Before the cardex system can be used though, the method of storage has to be defined. The simplest method of doing it is to create a class containing the primitive types. SQLite can only store certain types of data: `integer`, `real`, `text`, `none`, and `numeric`. No other types are permitted – this includes arrays and collections (such as `List<T>`). The types normally used in programming (such as `string` and `double`) are **mapped** to these internal types.

A table also requires a primary key. This is the main index key which is typically auto-incremented. In a data class, this would be defined using `[PrimaryKey, AutoIncrement]`.

A simple database class

As a demonstration, a simple database class can be used:

```
using SQLite;
public class demoRow
{
    public demoRow ()
    {
        [PrimaryKey, AutoIncrement]
        public int ID
        {get; set;}
        public string Name
        {get;set;}
        public double Value
        {get;set;}
        public override string ToString()
        {
            return string.Format("[demoRow : ID={0}, Name={1},
                Value={2}]", ID, Name, Value);
        }
    }
}
```

The `demoTable` variable can then be brought into the database.

Create a connection to the database

Before the database can be used, a connection to the server needs to be set up. As with any class, the database class needs to be set up. I've called mine `DataManager`. `SQLite` needs a path to the database file.

```
DataManager dm = new DataManager("path_to_database");
dm.Setup(); // calls the creation of the database code
```

The preceding code sets up an instance of the `DataManager`. To enable the database to be used across the app, the following should be added to the `AppDelegate` class:

```
private static DataManager dm
{ get; set; }
```

Within the `DataManager` class, a lock is required (in order to prevent more than one operation occurring on the database at any time), as is a local copy of the database path.

```
public DataManager(string path)
{
    dataLock = new object();
    dataBasePath = path;
}

private string dataBasePath;
private object dataLock;

public string DataPath
{
    get
    {
        return dataBasePath;
    }
}
```

Finally, the table needs to be set up in the database.

```
public bool Setup()
{
    lock(dbLock)
    {
        try
        {
            using (SQLiteConnection sqlCon = new
                SQLiteConnection(DBPath))
```



```
        {
            sqlCon.CreateTable<demoRow>();
        }
        return true;
    }
    catch (SQLiteException ex)
    {
        throw ex;
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
}
```

You'll notice that you will need to define a class with the following constants in it:

```
public const string DBClauseSyncOff = "PRAGMA SYNCHRONOUS=OFF;";
public const string DBClauseVacuum = "VACUUM;";
```

The `DBClauseVacuum` constant is used on the final `Execute` query. The `DBClauseSyncOff` constant is used on the first.

At this point, you may have noticed something about using SQLite. It is being used as if it is a normal method within a class. This is fine as it is.

Setting up an SQLite helper class

Typically using SQLite would require you to make and store the connection globally (to save device resources and reduce the possibility of a security problem) and then for every call to the database set up a set of queries and issues. Even for a trivial database, this can lead to many lines of repeated code and more the lines of code you have, the greater is the potential for bugs to crop up.

A helper class encapsulates all of the functionalities you will need and is very easy to write.

Writing helper class methods

As I mentioned at the outset, databases allow you to read/write from/to a table. To start with, it makes sense to be able to read data. The following are two classes: the first returns `List<demoRow>` and the other returns a name for the given ID.

```
Public List<demoRow> getAllListOfRows()
{
    lock (dbLock)
    {
        using (SQLiteConnection sqlCon = new
            SQLiteConnection(this.DBPath))
        {
            sqlCon.Execute(Constants.DBClauseSyncOff);
            sqlCon.BeginTransaction();
            List<demoRow> toReturn = new List<demoRow>();
            toReturn = sqlCon.Query<demoRow>("SELECT * FROM
                demoRow");
            return toReturn.Count != 0 ? toReturn : new
                List<demoRow>();
        }
    }
}

public string getNameForID(int id)
{
    lock (dbLock)
    {
        using (SQLiteConnection sqlCon = new
            SQLiteConnection(DBPath))
        {
            sqlCon.Execute(Constants.DBClauseSyncOff);
            sqlCon.BeginTransaction();
            string toReturn = string.Empty;
            toReturn = sqlCon.ExecuteScalar<string>("SELECT Name
                FROM demoRow WHERE ID=?", id);
            return !string.IsNullOrEmpty(toReturn) ?
                toReturn : "No name found";
        }
    }
}
```

There is very little difference between the two methods except for the database call. The List version requires a query – this is used when a non-primitive type is used and the data is expected back out. The `ExecuteScalar` method expects rows that are of primitive type to be returned.

Accessing these helper methods would be the same as accessing any other method.

```
string name = dm.getNameForID(3);
```

or

```
List<demoRow> dT = dm.getListOfTables();
```

Adding data to the database

Data addition comes in the form of inserting or updating, and unlike a read, data addition needs to be trapped in case the insert or update fails and the database needs to be rolled back to before the attempt to add data.

Thankfully, an insert or update can be handled in one method.

```
public void AddOrUpdateTable(demoRow dTRow)
{
    lock (dataLock)
    {
        using (SQLiteConnection sqlCon = new
            SQLiteConnection( DBPath))
        {
            sqlCon.Execute(Constants.DBClauseSyncOff);
            sqlCon.BeginTransaction();
            try
            {
                if (sqlCon.Execute("UPDATE dataRow SET " +
                    "ID=?, " +
                    "Name=?, " +
                    "Value=? WHERE " +
                    "ID=?",
                    dRow.ID,
                    dRow.Name,
                    dRow.Value,
                    dRow.ID) == 0)
                {
                    sqlCon.Insert(dRow, typeof(demoRow));
                }
            }
        }
    }
}
```


With this in mind, when using LINQ within an application, you may encounter random crashes – it is unlikely, but it may happen and it's worth mentioning it.

LINQ – a whistle-stop tour

For my examples, I'll use the `demoRow` table already used in this chapter.

```
List<demoRow> myRow = dm.getListOfRows();
```

Somewhere in the returned `List` there is the name `Fred Moriarty` and I want to get from `List` the instance of the class with that name in it. I know there is only one instance of this name in the list.

```
var demo = myRow.SingleOrDefault(t=>t.Name == "Fred Moriarty");
```

The preceding code takes the table and returns the single instance. If the name is not found, `null` (or the default value) is returned.

Say my list contains a number of `Fred Bloggs` instances in the `Name` field.

```
var bloggs = myTRow.Where(t=>t.Name == "Fred Bloggs").ToList();
```

Before LINQ came along, this would have been quite a slow affair and would have required a code such as follows:

```
List<demoRow> bloggs = new List<demoRow>();
foreach(demoRow blog in bloggs)
{
    if (blog.Name == "Fred Bloggs")
        bloggs.Add(blog);
}
```

The previous examples are very simple. LINQ can perform very complex operations as well, such as:

```
var res = from inviter in ContactList
          from tester in invites
          where inviter.UserID == tester.UserId
          select tester;
```

Here we have two lists (`ContactList` and `invites`). The LINQ query creates two loops and selects the instance `tester` when `UserID` from the outer loop matches `UserID` from the inner loop. The result is pretty much instant.

SELECT and WHERE in LINQ – a common cause of confusion

A very common error that at some point everyone meets with LINQ is mixing the WHERE syntax up with the SELECT syntax.

The WHERE syntax is a condition (for example, WHERE ID==311 or WHERE A==B). It only returns the condition set. In the following example, testClass is a list, it contains a class of which there is a string called Value.

```
var retString = testClass.Where(t=>t.Value == "fred").ToList();
```

The retString variable will contain a List<string> of all results from testString where Value == "fred".

The SELECT syntax returns something for all items in the object passed in it. The result might be the items themselves but can be something else.

```
var retString = inString.Select(t=>t.ToUpper()).ToList();
```

The preceding code transforms the contents of inString to be in upper case and is output to a list.

```
var retString = inString.Select(t=>t.Value, Func<inString, outString>).ToList();
```

Here Func<inString, outString> transforms the elements of inString to outString and outputs the element as an outString list.

Using Select in LINQ

Take the following example and remember Select performs a transform:

```
string[] teams = {"Liverpool", "Everton", "Oldham", "Leeds"};
var result = teams.Select(t=>t.ToUpper());
foreach (string team in result)
{
    Console.WriteLine(team);
}
```

This will be the output:

```
LIVERPOOL
EVERTON
OLDHAM
LEEDS.
```

But what is happening? The `Select` statement is performed on the string array `teams`. It then specifies a lambda expression (`t=>`) which in turn transforms the expression to be upper case (`ToUpper()`).

The `Select` statement (as has been seen in the previous code) has an overloaded method as well. Again, it is a transforming method.

```
string[] teams = {"Liverpool", "Everton", "Oldham", "Leeds"};
var trans = teams.Select(teams, index) =>
    new {index, str = teams.Substring(0, index)};
```

If the code was run and a suitable output was used, you would see `E, OL, Lee` – the index at the start is 0, so no characters are seen.

Replacing SQL with LINQ

Depending on your needs, it may be a better option to replace an SQLite database with a series of `List<T>` classes and use LINQ to replace the SQL queries. For example, if you have a very simple database (such as our `demoTable` database) which has a very limited scope for manipulation to be used, it may be a better idea to use a list of classes, add to them as you would normally do, and perform the queries using LINQ. Without the hit on the database server, this may yield a faster response time from the application.

For a more complex table structure where the SQL itself performs a `JOIN` to another table or there are complex manipulations involved, using LINQ may not result in a usable system.

LINQ can perform `JOIN` conditions as well as most other functions that SQLite can perform – just not as easily.

Remember though, LINQ on the iPhone may decide to just die whereas SQLite won't.

Summary

Data storage on the iPhone can be simple and can also be fraught with problems. For safety and reliability, the SQLite option is preferred on the iPhone. For speed, you can't beat LINQ but you must ensure that you test the LINQ project on a physical device when using LINQ.

12

Peripherals

The whole point of having a smartphone is that it is not just a phone, it's a GPS, a media center, a messenger system, and a video system. In short, that little device in your pocket is the proverbial "Jack of all trades, but master of *all*."

In this chapter, we will be covering the following topics:

- Using the camera
- Maps and GPS
- Storage on the phone
- Making a phone call
- Sending and receiving a text message
- Accessing the internet
- Multimedia

Using the camera

The camera on the iPhone is capable of recording stills and video. We will be dealing with video shortly.

The camera can be accessed in one of the two ways. Xamarin has released `Xamarin.Mobile` in its component store, which gives a cross platform method to access the GPS, camera, and address book. For completeness, we will cover both the native and component versions.

Accessing the camera (Xamarin.Mobile)

The `Xamarin.Mobile` component provides an easy way to access a camera. A simple method would look as follows:

```
var picker = new MediaPicker ();
if (!picker.IsCameraAvailable)
    Console.WriteLine ("No camera!");
else {
    picker.TakePhotoAsync (new StoreCameraMediaOptions {
        Name = "test.jpg",
        Directory = "MediaPickerSample"
    }).ContinueWith (t => {
        if (t.IsCanceled) {
            Console.WriteLine ("User canceled");
            return;
        }
        Console.WriteLine (t.Result.Path);
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

The issue with the `Xamarin` component is that it currently doesn't provide a method for accessing the front camera; however, the preceding code will save the clicked image.

Accessing the camera (Native)

The camera is accessed using `UIImagePickerController`. To use it, it's always wise to first check whether the device actually has a camera (the iPod Touch doesn't have one nor does the simulator). To do this, check the `IsSourceTypeAvailable` Boolean.

```
var myCamera = UIImagePickerControllerSourceType.Camera;
if (UIImagePickerController.IsSourceTypeAvailable(myCamera))
{
    UIImagePickerController myCameraPicker = new
        UIImagePickerController();
    myCameraPicker.SourceType = myCamera;
    myCameraPicker.Delegate = new myImageDelegate(this);
    PresentModalViewController(myCameraPicker, true);
}
```

The delegate deals with dismissing the modal window and any other process you want (such as displaying the picture or saving the image). A simple delegate would look something like the following (it can be extended if required):

```
public class myImageDelegate : UIImagePickerControllerControlDelegate
{
    private UIViewController myController;
    public myImageDelegate(UIViewController control) {
        myController = control;
    }
    public override void FinishedPickingMedia (
        UIImagePickerController thePicker, NSDictionary I) {
        myController.DismissModalViewControllerAnimated(true);
    }
}
```

Saving to the Photo album (Native)

The following code would be placed in the `myCameraPicker.Delegate` code within the `FinishedPickingMedia()` method, which would save images to the camera roll natively:

```
UIImage myImage = UIImage.FromFile(filename);
myImage.SaveToPhotoAlbum(delegate (UIImage image, NSError err) {
    Console.WriteLine("Image saved fine"); });
```

GPS and Mapping

This is also covered by the `Xamarin.Mobile` component in part (the full functionality of **Core Location** is not replicated in the `Xamarin.Mobile` component, due to there not being an equivalent method on the other platforms that the component supports). Thankfully, the component and `Core Location` work together seamlessly.

GPS with Xamarin.Mobile

The `Xamarin.Mobile` component allows you to listen to the position of the device and act when the position has been changed, using the `PositionChanged` event. The following code demonstrates how to use it:

```
var iPhoneLocationManager = new Geolocator();
iPhoneLocationManager.DesiredAccuracy = 5;
```

```
iPhoneLocationManager.StartListening(1, 10);
iPhoneLocationManager.PositionChanged += (object sender,
    PositionEventArgs e) => {
    double geoLocationLong = e.Position.Longitude;
    double geoLocationLat = e.Position.Latitude;
    iPhoneLocationManager.StopListening();
};
```

While analyzing the preceding lines of code, we come across the following terms:

- **DesiredAccuracy:** It is the distance in meters that is needed before the event is triggered. This in itself can cause an issue. Anything under that value will mean that the event is not triggered. Too big a value, the accuracy is hit.
- **StartListening:** It takes two parameters: `minimumTime` and `minDistance`. In this case, it should either return within the first minute, or if the phone is moved more than 10 m.
- **StopListening:** It stops the listening service.

The `Xamarin.Mobile` module also provides you with an asynchronous method—`GetPositionAsync`, which retrieves your position asynchronously.

```
private TaskScheduler sched =
    TaskScheduler.FromCurrentSynchronizationContext();
private CancellationTokenSource cancel;
geolocator.GetPositionAsync (timeout: 10000,
    cancelToken: cancel.Token, includeHeading: true).ContinueWith (
    t=> {
    if (t.IsFaulted)
        Console.WriteLine("Position faulted : {0}").((
            GeolocationException)t.Exception.InnerException).Error);
    else if (t.IsCanceled)
        Console.WriteLine("Canceled");
    else {
        Console.WriteLine("Timestamp {0}",
            t.Result.Timestamp.ToString("G"));
        Console.WriteLine("La: {0}", t.Result.Latitude.ToString(
            "N4"));
        Console.WriteLine("Lo: {0}", t.Result.Longitude.ToString(
            "N4"));
    }
}, sched);
```

The returned value stored in `Result` also provides the speed. This is not to be relied upon. It's far more accurate to use your own method, but to do that you need to know how far you have traveled.

Calculating your speed

The `CLLocation` namespace has in it a method named `DistanceFrom` for calculating the distance. The method works as follows:

1. Gets the new location.
2. Gets the old location.
3. Creates two instances of `CLLocationManager`.
4. Puts the coordinates of both the locations into the two instances respectively.
5. Uses `DistanceFrom`.

The resulting `double` value gives you the distance in meters and the time in seconds, so the speed you have traveled is in meters/seconds (that is, distance/time). The issue here though is the calculation; it might consider how a bird flies rather than how you are travelling. For example, if the desired accuracy you have is too low and the timeout too high, the method with `DistanceFrom` will still work; however, if the distance is not 100 m, but apparently 35 m (consider that a bird flies straight across a field, and does not walk down a road, turn left, turn right, or go roundabout and then turn left again), the distance is much shorter and therefore will also change the speed considerably.

Using Core Location

Core Location is the default framework for the GPS. It is a powerful system for determining the positioning of the device and is far reaching in what it does and how it does it. As with a number of other facilities within iOS, a delegate is required when setting up the Core Location framework. A delegate typically handles events.

Setting up Core Location and delegate

Core Location and delegate can be set up and used as demonstrated with the following code:

```
private CLLocationManager locationManager;
public override void ViewDidLoad() {
    base.ViewDidLoad();
    locationManager = new CLLocationManager() {
        DesiredAccuracy = CLLocation.AccuracyBest,
    };
}
```

CLLocationManager is set up with the desired accuracy being set to the best it can. Next, to make it useful, we need to first catch any errors with the following code:

```
locManager.Failed += (object sender, NSErrorEventArgs e) => {
    UIAlertView alert = new UIAlertView() {
        Title = "Location manager failed",
        Message = string.Format("The following error was encountered -
        {0}", e.Error.ToString())
    }.Show();
    locManager.StopMonitoring();
};
```

Once the error system has been set, monitoring the changes to the positioning needs to be handled.

```
locManager.LocationsUpdated += (object sender,
    CLLocationManager.LocationsUpdatedEventArgs e) => {
    CLLocation[] locs = e.Locations;
    Console.WriteLine("lat = {0},
        long = {1}",
        locs[0].Coordinate.Latitude,
        locs[0].Coordinate.Longitude);
};
```

If the app is paused, this event will need to be handled:

```
bool paused = false;
locManager.LocationUpdatesPaused += delegate {
    if (!paused)
        locManager.StopUpdatingLocation();
    else
        locManager.StartUpdatingLocation();
    paused = !paused;
};
```

To start the checks for update monitoring, use the following line of code:

```
locManager.StartUpdatingLocation();
```

To stop the checks for update monitoring, use the following line of code:

```
locManager.StopUpdatingLocation();
```

The `CLLocationManager` class also allows you to monitor the direction you are moving in.

```
locManager.UpdatedHeading += (object sender,
    CLHeadingUpdatedEventArgs e) => {
    CLHeading heading = e.NewHeading.TrueHeading;
    Console.WriteLine("New heading : {0}", heading.ToString());
};
locManager.StartUpdatingHeading();
```

Finding where the user is

Having coordinates is all well and good, but who can actually say what they mean? For example, if I were to give you the longitude/latitude coordinates of 53.431/-2.956, would you know where that was? Chances are that you wouldn't!

This is where reverse geocoding comes into the picture. The `MKReverseGeocoder` class is in the `MonoTouch.MapKit` namespace.

```
private MKReverseGeocoder geoCoder;
geoCoder = new MKReverseGeocoder(locManager.Location.Coordinate);
geoCoder.Delegate = new ReverseGeocoder(this);
geoCoder.Start(); // geoCoder.Stop() stops the geoCoder
```

Unlike `locManager`, the `ReverseGeocoder` class has to be placed into a delegate. `MKReverseGeocoder` doesn't contain any events to latch on to.

```
private class ReverseGeocoder : MKReverseGeocoderDelegate {
    UIViewController view;
    public ReverseGeocoder(UIViewController myView) {
        view = myView;
    }

    public override void FoundWithPlacemark(
        MKReverseGeocoder geocoder, MKPlacemark placemark) {
        Console.WriteLine("Address");
        Console.WriteLine("{0}, {1}, {2}, {3}, {4}",
            placemark.SubThoroughfare,
            placemark.Thoroughfare,
            placemark.Locality,
            placemark.AdministrativeArea,
            placemark.Country);
    }
}
```

The `Xamarin.Mobile` component does not contain the ability to perform a `ReverseGeocoder` operation. But it's not difficult to write one.

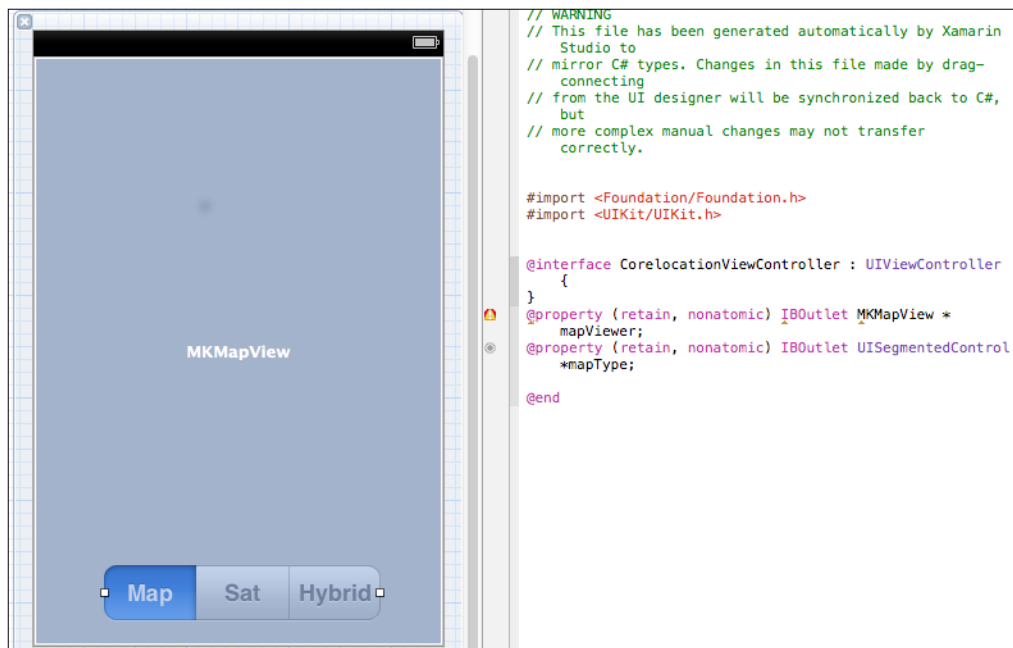
In the original example, the `LocationsUpdated` method was given as an inline example. If this is changed to point to a method, the `ReverseGeolocation` method can be performed:

```
private void OnPositionChanged(object sender,
    PositionEventArgs e) {
    double lng = e.Position.Longitude;
    double lat = e.Position.Latitude;
    CLLocation clloc = new CLLocation(lat, lng);
    CLLocation oldLoc = new CLLocation(prevLat,prevLong);
    CLGeocoder geoRevGeo = new CLGeocoder();
    geoRevGeo.ReverseGeocodeLocation(clloc, GetAddressFromLoc);
}

private void GetAddressFromLoc(CLPacemark[] place,
    NSError error) {
    if (place.Length == 0) {
        Console.WriteLine("Don't know where I am - help!");
        return;
    }
    else {
        Console.WriteLine("Address");
        Console.WriteLine("{0}, {1}, {2}, {3}, {4}",
            place.SubThoroughfare,
            place.Thoroughfare,
            placemark.Locality,
            placemark.AdministrativeArea,
            placemark.Country);
    }
}
```

Adding a map

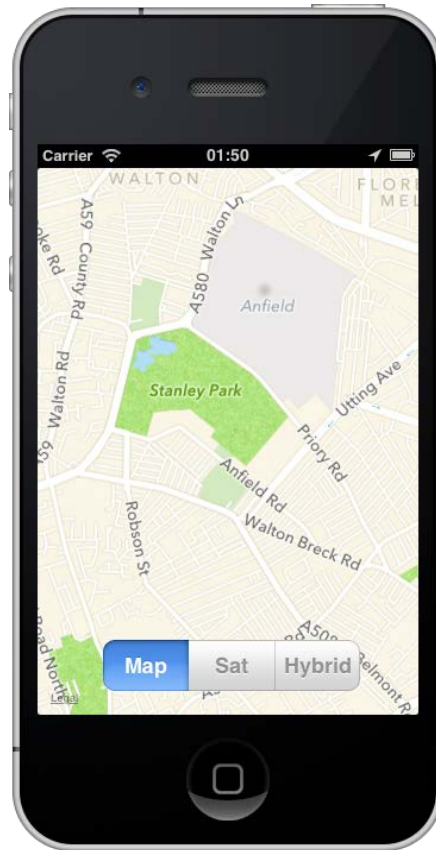
The final part to the GPS is to add a map. For this, a Map view needs to be used by adding it in either Xcode or code. For my purpose, I will assume it was added via Xcode and is called `mapViewer`. There are three types of maps available, so a `UISegmentedControl` is also added.



The `mapViewer` object needs to be set up:

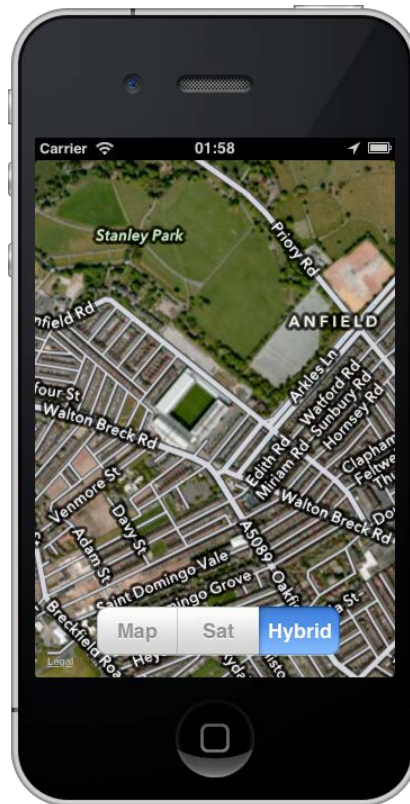
```
public override void ViewDidLoad() {
    base.ViewDidLoad();
    mapViewer.ShowsUserLocation = mapViewer.ZoomEnabled = true;
    mapViewer.MapType = MKMapType.Standard;
    mapViewer.Region = new MKCoordinateRegion(
        new CLLocationCoordinate2D(53.431, -2.956),
        new MKCoordinateSpan(0.5, 0.5)
    );
    mapViewer.ScrollEnabled = mapViewer.UserInteractionEnabled =
        true;
}
```


The resultant location corresponding to the coordinates passed in the preceding code is shown in the following screenshot:



The next stage is to make the UISegment control go live:

```
mapType.ValueChanged += delegate {
    switch (mapType.SelectedSegment) {
        case 0: mapView.MapType = MKMapType.Standard;
                break;
        case 1: mapView.MapType = MKMapType.Satellite;
                break;
        case 2: mapView.MapType = MKMapType.Hybrid;
                break;
    }
};
```



In case you didn't know, the geolocation is for **Anfield**, home of Liverpool FC.

Some of the properties need to be explained in short:

- `ShowUserLocation`: It shows a blue dot to show where the user is
- `ZoomEnabled`: It allows the user to zoom in
- `ScrollEnabled`: It allows the user to scroll the view around
- `UserInteractionEnabled`: It enables a pin placed on the map to respond (or not) when clicked
- `MKCoordinateSpan(0.5, 0.5)`: This is the zoom setting – the smaller the number, the larger the zoom on the initial view

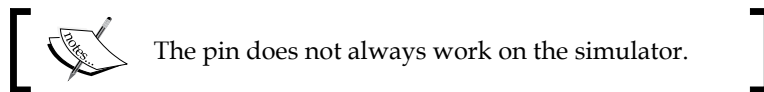
Adding a pin

When it comes to maps, *pins* are very useful. While a little blue dot is neat, a pin really shows you where you are, and information can be added to it.

```
MKUserLocation loc = new MKUserLocation() {
    Title = "Anfield stadium",
    Coordinate = new CLLocationCoordinate2D(53.431, -2.956),
    Subtitle = "Home of Liverpool FC",
};
mapViewer.AddAnnotationObject(loc);
```

Finally, it is always good to center the map and pin on the screen

```
mapViewer.SetCenterCoordinate(loc.Coordinate, true);
```



Storage on the phone

.NET specifies a number of places in the `SpecialFolders` enumeration (such as `Program Files`, `My Music`, and `My Pictures`; a full list can be found on the Microsoft website). Due to security restrictions on the iOS devices, only a few of them are available. It is safest to restrict saving any user data to `My Documents`. Within `My Documents`, you can create directories of your own and use them the way you like.

Making a phone call

This may sound daft. Why would you want to write code to make a phone call? In true developer tradition, the answer is *why not?* It is important to note that a string number is just that. It cannot contain spaces, hyphens, brackets, or the plus sign (+); it can only contain numbers.

```
private void callNumber(string number) {
    string phoneURLString = string.Format("tel:{0}", number);
    NSURL phoneURL = new NSURL(phoneURLString);
    UIApplication.SharedApplication.OpenUrl(phoneURL);
}
```

Moving on...

Sending and receiving a text message

The iPhone comes with its own built-in messaging software. However, there are some rare times when you need to code a message before sending it. Apple, in its wisdom, though, does not allow you to send a message without it going through its own message software. This is not to say you can't send a message in another way (such as through a dedicated web or message service), but for standard users, you can't.

You cannot code in a way that would intercept or block text messages. iOS provides no publicly available method to intercept and read a text message. With that in mind, sending a text message is a relatively straightforward affair. `MFMMessageComposeViewController` is in the `MonoTouch.MessageUI` namespace.

```
private void sendTextMessage(string number, double myLat,
double myLong) {
    if (MFMMessageComposeViewController.CanSendText) {
        MFMMessageComposeViewController message = new
            MFMMessageComposeViewController();
        message.MessageComposeDelegate = new
            CustomMessageComposeDelegate();
        message.Recipients = new string[] { number };
        message.Body = string.Format("Help! I am currently at
            https://maps.google.com/maps?q={0},{1}&z=18 and need
            assistance", myLat, myLong);
        NavigationController.PresentModalViewController(message,
            true);
    }
}

public class CustomMessageComposeDelegate :
    MFMMessageComposeViewControllerDelegate {
    public override void Finished(
        MFMMessageComposeViewController controller,
        MessageComposeResult result) {
        if (result == MessageComposeResult.Failed ||
            result == MessageComposeResult.Cancelled) {
            UIAlertView alert = new UIAlertView() {
                Title="Message sending error",
                Message="Your message failed to send"
            }.Show();
        }
        controller.DismissViewController(true, null);
    }
}
```

Accessing the Internet

Access to the internet is via the `UIWebView` controller. Prior to trying to access an internet site, it's a good idea to ensure there is a live network connection. This is preformed via the `NetworkReachability` class.

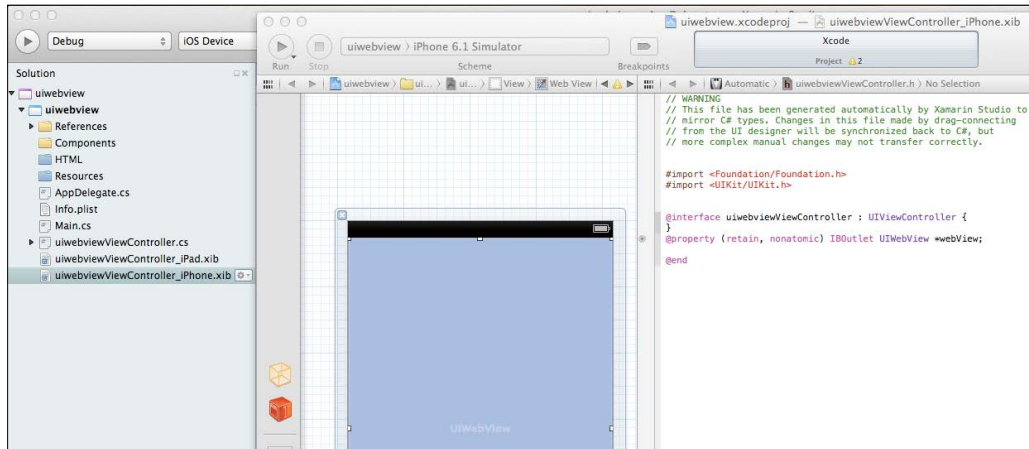
```
using MonoTouch.SystemConfiguration;
NetworkReachability reach = new NetworkReachability(
    "www.bbc.co.uk");
NetworkReachabilityFlags flags = new NetworkReachabilityFlags();
reach.TryGetFlags(out flags);
Console.WriteLine("Network flag = {0}", flags.ToString());
```

The `WriteLine` command will result in one of the following flags:

Flag	Value
<code>Reachable</code>	The host is reachable
<code>ISWWAN</code>	Connection is made through EDGE, 3G, or 4G
<code>IsLocalDevice</code>	Connection is made to the local device
<code>ConnectionAutomatic</code>	Connection is made automatically. This is an alias for <code>ConnectionOnTraffic</code> .
<code>ConnectionOnTraffic</code>	A combination of <code>Reachable</code> and when data is requested, a connection is made
<code>ConnectionOnDemand</code>	Occurs when the connection starts. The connection occurs once the socket is connected.
<code>ConnectionRequired</code>	The host can be reached, but the connection must be made
<code>IsDirect</code>	The connection is made directly
<code>InterventionRequired</code>	When connected to a host, the user must do something
<code>TransientConnection</code>	The host can be reached, but through a system which starts and stops (such as PPP or any other nonpersistent network connection)

Assuming there is a network connection, the next stage is to load the web page.

I covered loading a web page and some of the settings back in *Chapter 3, Views and Layouts*. It is also possible to load a web page from the data you have dynamically generated. This can be simply demonstrated using the following UI with the code. Firstly, set up an application structure as shown with `UIWebView` in the `.xib` file:



The HTML directory can be named as per your wish. I've kept it simple and just called it HTML. When generating your own HTML from within your app, you can either:

- Generate your own file from scratch, save, and get the output
- Generate your HTML file using `StringBuilder` and output that string
- Mix the preceding two options

In essence, the two generation methods are the same with the difference of generating the string either using `StringBuilder` or concatenate strings. Consider the following lines of code:

```
string html = "<html>\n";
html += "<title>Hello World</title>\n";
html += "<body>\n";
html += "<h1>Hello World!</h1>\n";
html += "</body>\n";
html += "</html>";
```

The preceding lines of code should do what you would expect them to, once passed to `UIWebView` using the following code:

```
webView.LoadHTMLString(string s, NSURL baseurl);
//The baseurl here would be null.
```

While these lines of code do their work, creating your web page based on data from within an app using these lines of code can be time consuming. A simpler method is to pull the header and footer in from some HTML fragments within the app.

To read the files from within the app, though, a couple of steps have to be taken. The first is to set the HTML fragments to be built as `BundleResource`. You will be loading a file that is part of the bundle rather than a file in the app's writable folder. The second part is to load the HTML bundles into the source:

```
var documents = NSBundle.MainBundle.BundlePath;
StringBuilder sb = new StringBuilder();
sb.Append(File.ReadAllText(Path.Combine(documents,
    "HTML/top.html")));
```

`NSBundle.MainBundle.BundlePath` is the path to the installed application. The next step is to add the data; in my example, I am adding a league table:

```
var leagues = (from t in teams
               from p in t.points
               orderby p
               select t).Take(4).ToList(); // takes top 4 only
sb.Append(@"<table width=100%>");
sb.Append(@"<tr width=100%>");
sb.Append(@"<td width=70%>Team name</td>");
sb.Append(@"<td width=10%>Played</td>");
sb.Append(@"<td width=10%>G Diff</td>");
sb.Append(@"<td width=10%>Points</td>");
for (int i = 0; i < 4; ++i) {
    sb.Append(@"<tr width=100%>");
    sb.Append(@"<td width=30%>" + teams[i].TeamName + "</td>");
    sb.Append(@"<td width=10%>" + teams[i].TeamPlayed.ToString() +
        "</td>");
    sb.Append(@"<td width=10%>" + teams[i].TeamGDiff.ToString() +
        "</td>");
    sb.Append(@"<td width=10%>" + teams[i].TeamPts.ToString() +
        "</td>");
    sb.Append(@"</tr>");
}
sb.Append(@"</table>");
sb.Append(File.ReadAllText(Path.Combine(documents,
    "HTML/bottom.html")));
```

The top table section could as easily be its own HTML fragment. Once the final append has been made, the HTML is good to go.

```
webView.LoadHTMLString(sb.ToString(), null);
```

Remember, what you're doing is creating your own web page within the application. If you want to include stylesheets, you can. If you want to include JavaScript, you can; the only caveat is that you can use mobile Safari, which has JavaScript enabled.

Multimedia

This is a very vast topic. Thankfully, using a camera has been covered at the start of this chapter. Just about everything you need is inside the `MonoTouch.MediaPlayer` namespace. As with using a camera, it is important that you first check that the device has video capabilities in exactly the same way as you do for a camera (in fact, it uses the same `IsSourceTypeAvailable(myCamera)` command as used for a camera).

Playing a video

As with `webView`, a video can be external to the device or internal; if it is internal, it could be part of the bundle (not a good idea as video takes up a lot of space on a device, which will result in long download times), or within the `My Document` area (downloaded), or in the photo reel.

External URL

The video that I'll use here is on YouTube (you can choose any video of your choice).

```
var videoPlayer = new MPMoviePlayerController(  
    NSUrl.FromString("http://www.youtube.com/watch?v=cVikZ8Oe_XA"));  
videoPlayer.Play();
```

Internal source

Playing from an internal source can be performed in a way similar to that of an external video file:

```
var videoPlayer = new MPMoviePlayerController(  
    NSUrl.FromFilename("myVideo.mp4"));  
videoPlayer.Play();
```


From the photo library

Choosing a video from the camera roll is performed in much the same way as picking an image from the photo library.

```
UIImagePickerController ipcPicker = new UIImagePickerController()
{
    SourceType = UIImagePickerControllerSourceType.PhotoLibrary,
    MediaTypes = new [] {"public.movie"},
    Delegate = new ImagePickerDelegate(this)
};
```

The preceding code goes through the photo library looking for any file that returns the `public.movie` type. If found, the file is added to the array and can be seen via the delegate.

Recording a video

This is not substantially different from taking a picture, except that you have a few additional parameters that can be altered, such as `VideoQuality` and `AllowEditing`.

To record a video

Recording a video is a simple task as the following code demonstrates:

```
var camera = UIImagePickerControllerSourceType.Camera;
UIImagePickerController ipcVideo = new UIImagePickerController()
{
    SourceType = camera,
    MediaTypes = new [] {"movies.public"},
    AllowEditing = true,
    VideoQuality = UIImagePickerControllerQualityType.Medium,
    Delegate = new ImagePickerDelegate(this)
};
```

In the preceding example, `AllowEditing` has been set to `true`, which means that the user may edit this video. If that is the intention, editing should be performed using `UIVideoEditorController`. This controller allows for three events: `Failed` (the edit failed for some reason), `UserCancelled` (speaks for itself), and `Saved` (the user has selected to save the edit; the path is returned in the `e.Path` event).

Saving the video

Once the video has been processed, the next step is to save the video:

```
UIVideo.SaveToPhotoAlbum(videoPath, delegate (string path,
    NSError error) {
    Console.WriteLine("Video saved.");
});
```

The audio system

The iPhone and iPad range of devices is blessed (as are most Apple devices) with a fantastic audio system that allows for great playback quality and the ability to record as well. These facilities are available through the `AVAudioPlayer` class or `SystemSound`. If the file is held within the application bundle, it must be set to `Content` when building the app.

Playback

Audio playback can be considered short or long if it is under or over the 30 seconds mark. In general, `SystemSound` is best used for audio files with a duration of less than 30 seconds and also for uncompressed audio formats, such as `.wav` and `.caf` (**Core Audio File**). MP3 files are not supported in `SystemSound`.

Short files

The `SystemSound` method is a quick and easy way to play an audio file with very little overhead.

```
var sound = SystemSound.FromFile("myAudio.caf");
sound.PlaySystemSound();
```

If you need an audio file to play but you are at some place requiring silence (say, a library), the device can be made to vibrate through the length of the file.

```
sound.Vibrate.PlaySystemSound();
```

Long (and compressed) files

Here the `AVAudioPlayer` class comes into its own, allowing you to alter power levels (the volume on a channel) effectively, pause, play, and stop an audio file. It also handles compressed audio formats, such as MP3.

Setting the power levels

Prior to setting a power level, either or both the `AveragePower` and `PeakPower`. `MeteringEnabled` Booleans have to be set to `true` and a method named `UpdateMeters()` must be called. It is then just a case of setting `AveragePower(uint)` or `PeakPower(uint)` to the value you want (in dBs).

Playing the audio file

It is not difficult to play an audio file; select the file and tell the device to play. The following code demonstrates how:

```
var fileToPlay = AVAudioPlayer.FromUrl(NSUrl.FromFilename("myAudio.mp3"));
fileToPlay.Play();
fileToPlay.FinishedPlaying += delegate {
    fileToPlay.Dispose(); // clean up
};
```

Altering the volume

If you don't want Beethoven's ninth blasting out of your iOS device, it's a good idea to turn down the volume as follows:

```
using MonoTouch.MediaPlayer
var mpPlayer = new MPMusicPlayerController();
mPlayer.Volume = 0.01f; // max volume - range 0 to 1
```

A word of caution, though, for setting a volume level: *don't set it to zero*. This annoys the iOS device and will then annoy the user as the device will take great pleasure in reminding you that the volume is set to 0. As the figure is a float value, 0.01 will do just as well as 0 to mute the device.

Recording Audio

Recording is not as simple as playing. While a lot of the work is done by the `AVAudioRecorder` class, quite a bit of work also has to be done by the programmer to record the audio. The key point to remember is that `NSDictionary` needs to be set up before anything can happen. This dictionary contains important information, such as the type of audio, sample rate, quality, and so on.

Setting up the audio NSDictionary

The NSDictionary (along with anything else that starts with NS) is an interface to the Objective-C bindings that Xamarin.iOS utilizes to allow development with the .NET framework on iOS and, as such, can't be set up like a normal dictionary. To get around that obstacle, a generic NSObject object can be used, one for the settings, the other for the description (which works out to be the same as the values and keys in .NET).

```
var settings = new NSObject[] {
    NSNumber.FromFloat(22050.0f),
    NSNumber.FromInt32((int)AudioFileType.WAVE),
    NSNumber.FromInt32(2),
    NSNumber.FromInt32((int)AVAudioQuality.Min)
};
var keysToSettings = new NSObject[] {
    AVAudioSettings.AVSampleRateKey,
    AVAudioSettings.AVFormatKey,
    AVAudioSettings.AVNumberOfChannelsKey,
    AVAudioSettings.AVEncoderAudioQualityKey
};
var dict = NSDictionary.FromObjectsAndKeys(settings,
    keysToSettings);
```

Setting up to record

The next step is to set up the recorder itself and, importantly, the location to save the audio file to.

```
var docs = Environment.GetFolderPath(
    Environment.SpecialFolder.Personal);
var audio = NSUrl.FromFilename(Path.Combine(docs,
    "testaudio.wav"));
var error = new NSError(); // catch the errors
```

Recording the audio file

Finally, it's time to record. As the file records, it's also saved to the device. Once finished with recording, the recorder object needs to be disposed. Thankfully, we can control how long a recording goes on by using the RecordFor("float time") method.

```
var myRecorder = AVAudioRecorder.ToUrl(audio, dict, out error);
myRecorder.FinishedRecording += delegate {
    Console.WriteLine("Audio file created");
    myRecorder.Dispose();
};
myRecorder.RecordFor(10f);
```

Summary

It is simple enough to use the subsystems that the iOS devices make use of, as long as you remember the limitations, such as only being able to send texts and not receive. The devices offer far more than what the average user sees and it only requires a small amount of imagination to see how to create a really good application using the facilities offered to you; from maps to web views to making calls, it's all there for you. Now play!

13

User Preferences

From time to time, it is useful for a user to be able to store preferences (such as ringtones for a contact). Thankfully, iOS provides a built-in preference system. This is fine if you only ever wish to write an app for iOS, but if the intention is to have an app that can be built for any platform that can use .NET, then a different approach is required.

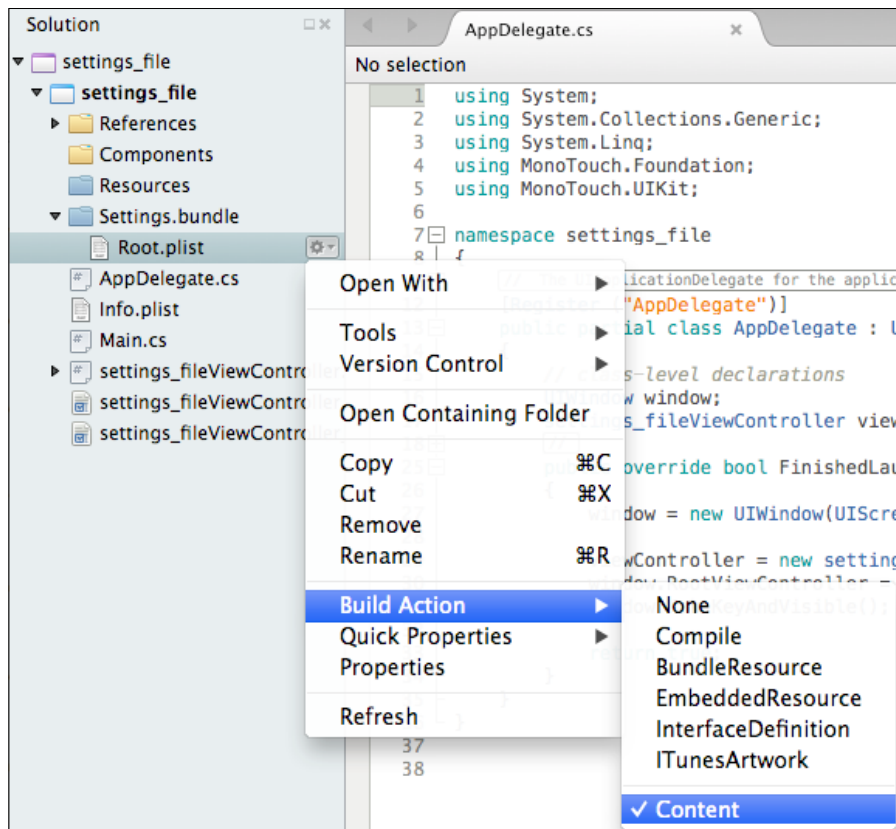
The topics covered in this chapter will be:

- Using the built-in preference system
- Rolling your own preference system

The built-in system

To start with, a special folder has to be added to your application called `Settings.bundle` that contains a file called `Root.plist`.

This is not a normal file and the `.plist` file needs **Build Action** to be set as **Content**.



The `.plist` file is an XML file with entries stored in a regular dictionary objecting the `<key><type>` form, where type can be anything (such as `string`, `int`, and `double`).

For example, for creating a preference displayed as a `UISlider` class it would be stored as shown in the following code:

```
<dict>
  <key>Type</key>
```

```

    <string>UISlider</string>
    <key>Key</key>
    <string>My slider</string>
    <key>DefaultValue</key>
    <int>40</int>
    <key>MinimumValue</key>
    <int>0</int>
    <key>MaximumValue</key>
    <int>75</int>
</dict>

```

It is simple to understand, but the file can grow quickly, depending on how much information is being stored.

Reading and writing to the .plist file

Thankfully, it is simple to read from the .plist file. All that is required is the key to identify the value and the knowledge of the value type to properly call the related method.

```

int read =NSUserDefaults.StandardDefaults.IntForKey
("DefaultValue");
string name = NSUserDefaults.StandardDefaults.StringForKey("Key");

```

Writing to the file can be performed in two ways. The simplest is to just write to the file, as shown in the following code:

```

NSUserDefaults.StandardDefaults["DefaultValue"] = 34;
NSUserDefaults.StandardDefaults["Key"] = "Hello mum!";

```

The other way is to call the `NSNotificationCenter` class, which broadcasts notifications to the application. The `NSUserDefaults` class uses this system to emit the `NSUserDefaultsDidChangeNotification` notification when the `Settings` values change. The good part here is that any `NSObject` class can be set to act as an observer for the notification. The observer provides a call back method as shown in the following code:

```

public mySettingsClass()
{
    NSNotificationCenter.DefaultCenter.AddObserver
    (
        this, new Selector("updateSettings"),
        new NSString("NSUserDefaultsDidChangeNotification"),
        null
    )
}

```



```
    );

    [Export ("updateSettings:")]
    private void UpdateSettings()
    {
        doSomething();
    }

    private void doSomething()
    {
        // do something here
    }
}
```

Rolling your own settings system

While using the built-in settings system may seem useful, the point of using the Xamarin range of products is to be able to use a large amount of the same code on any platform that now supports .NET. While, at the time of writing, Windows Mobile is languishing at around three percent of the market share of all smart phones, Microsoft is unlikely to allow this to continue and will push their massive reserves into getting people to adopt their smart phones. If you want the pure mathematics, 97 percent of all smart phones can be coded using the .NET platform. (Blackberry has a port of mono for it, but it is not well supported and so can be discounted.)

This, therefore, requires a different strategy for storing user settings. We could use an SQLite database to store the details, but, as has been pointed out, each access to the sub system will cause a performance hit.

The simplest way is to create a settings class and **serialize** or **deserialize** the values as and when required. It is an extremely flexible approach and works wonderfully.

Serializing and deserializing data

The following code is a very simple serializer and deserializer:

```
public class Serializer
{
    public static void XmlSerializeObject<T>
        (T obj, string filePath)
    {
        using (StreamWriter sw = new StreamWriter(filePath))
        {
```

```

        XmlSerializer xmlSer = new XmlSerializer(typeof(T));
        xmlSer.Serialize(sw, obj);
    }
}

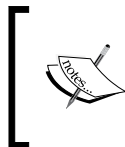
public static T XmlDeserializeObject<T>(string filePath)
{
    using (StreamReader sr = new StreamReader(filePath))
    {
        XmlSerializer xmlSer = new XmlSerializer(typeof(T));
        return (T)xmlSer.Deserialize(sr);
    }
}
}

```

The preceding code will serialize and deserialize any object type that is put through it. This includes generic types such as `List<>` and `Dictionary<, >` objects. Do not pass in interface members or circular references (like an object that refers to an object that refers to the original object) – this will cause the serializer to crash.

Setting up the Settings file

Unlike using the `NSUserDefaults.plist` file, the settings file used here is still an XML file, but it doesn't need to be set as any particular type. As it is also a simple XML file, it can be copied and transferred to the same application on a different platform.



So say you're configuring a file for the app on the iPhone and you want to run it on your Android phone as well. You will copy the `Settings` file and run the app on the Android phone and you instantly have the settings you had on the iPhone.

The `Settings` file consists of two classes – the handler class and the data class.

The handler class

The handler class handles setting up the `Settings` file as well as the accessors for the data class. I've reproduced the important parts in the following code:

```

public static class MyAppData
{
    private static AppSettings appSetting;
}

```

```
public static AppSettings appSettings
{
    get
    {
        if (appSetting == null)
        {
            if (File.Exists(AppSettingsFile))
                appSetting = Serializer.XmlDeserializeObject
                    <AppSettings>(AppSettingsFile);
            else
            {
                appSetting = new myAppData.AppSettings();
                Serializer.XmlSerializeObject <AppSettings>
                    (appSetting, AppSettingsFile);
            }
        }
        return appSetting;
    }
    set
    {
        if (value == null)
        {
            throw new ArgumentNullException
                ("value is null!");
        }
        appSetting = value;
        if (File.Exists(AppSettingsFile))
            File.Delete(AppSettingsFile);
        Serializer.XmlSerializeObject <AppSettings>
            (appSetting, AppSettingsFile);
    }
}

private static string pathAppSettingsFile;
public static string AppSettingsFile
{
    get
    {
        if (string.IsNullOrEmpty(pathAppSettingsFile))
            pathAppSettingsFile =
                Path.Combine(Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments),
                    "AppSettings.xml");
        return pathAppSettingsFile;
    }
}
}
```

Adding an accessor

The following code is used for any type of object being passed into the configuration file:

```
public static List<string> theAlias
{
    get { return appSettings.TheAlias;}
    set
    {
        AppSettings settings = appSettings;
        settings.TheAlias = value;
        appSettings = settings;
    }
}
```

The preceding simple accessor can be cut and pasted as many times as required. The best thing is that the `List<>` object can be of any type – you can even have an entire list of classes in there!

The data class

The data class contains nothing except a list of accessors that marry up to the ones in the data handler class, and its also contains a default constructor as shown in the following code:

```
public class AppSettings
{
    public AppSettings()
    {
    }

    public List<string> TheAlias
    {get;set;}
}
```

With the preceding code, you have possibly the most flexible configuration and settings system available.

Summary

User settings are important – who in their right mind wants to have to set up everything time and again, or worse, for every update of the application? When updating an application, a file within the application folder will be kept and will always be available (except if the app is uninstalled). Stored user settings are not always guaranteed to persist between versions. We have seen in this chapter two completely different methods of storing configuration settings: the one supplied and the one created. There are other ways to do the same as I have demonstrated in this chapter.

14

Testing and Publishing

Your app is ready and waiting to be released to the entire world. You've sat there, coded, tested, coded, tested, and finally decided that you've created the most amazing app in the world and it's time to get it onto the Apple store. The problem is that only you have tested it. Unlike testing on Android, testing on iOS is not as straightforward and neither is getting your app into the store.

In this chapter we will cover:

- Provisioning and signing your app
- Testing your app using TestFlight
- Packaging and signing the app
- Releasing it on the App Store

Provisioning and signing your app

For normal debugging to your own phone, it isn't usually required for you to sign the application because a generic developer code is used when you deploy to the phone. This code is known as a **provisioning profile**. We're now moving into a bigger league and the app needs testing. You may wonder why your app needs to be tested by other people. The answer is easy: to ensure the best possible user experience.

From the developer's perspective, a piece of software follows a set pattern: A goes to B, B goes to C, C goes to D or E, and so on. When we test our code, this is the path we take; users don't. They will go from A to C to F to B to C again and finally shout when the code takes exception and crashes for no reason while they try to go to H directly from C. The more people test your software before the release, the happier users are when they download the app from the **Mac App Store** and the fewer negative reviews get lodged against you.

For testing, you have to distribute the app, and it's not as simple as uploading the app to your personal web space and saying, "Hey, go download and install" as you would for a typical desktop (or Android) application. Apple, to ensure the quality of the available apps, has a fairly strict and tight system for distribution, even for testing. Luckily, there is a way to distribute your app prior to releasing it on the unsuspecting public: *TestFlight*.

TestFlight

TestFlight is a platform that allows you to upload the test (or beta) versions of your software on invited users (iOS devices) or registered iOS devices. It is a free service that does not break any of Apple's rules or regulations and, moreover, does not require the device to be jailbroken. Setting up this service is a three-step process:

- Provisioning
- Inviting
- Building and uploading

Provisioning

Provisioning sets up the device to allow for the testing of non-Apple store approved applications. To start with, you need to set yourself up as an iOS developer at <http://developer.apple.com>—the cost is around \$100 per year (about £65 at the time of writing). This is a yearly cost and you do get quite a bit for it. Once done, you will need to sign in and create an ID for your app.



Registering the app

First, select **Identifiers**: you need to give your app a name. In my example, given as follows, I have filled in both the bundle name and the app name using a unique identifier – it's always a good idea to keep your apps apart. You can set up a "catch all" identifier here if you wish.

ID Registering an App ID

The App ID string contains two parts separated by a period (.)—an App ID Prefix that is defined as your Team ID by default and an App ID Suffix that is defined as a Bundle ID search string. Each part of an App ID has different and important uses for your app. [Learn More](#)

App ID Description

Name:
You cannot use special characters such as @, &, *, ', ''

App ID Prefix

Value: (Team ID)

App ID Suffix

Explicit App ID

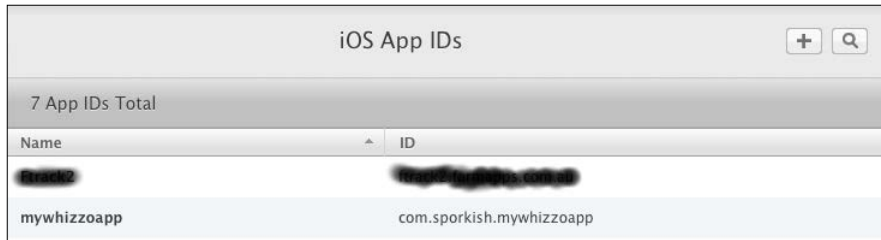
If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:
We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

The grayed out area (in the preceding screenshot) will be the ID that Apple will provide you when you first create an account with them.

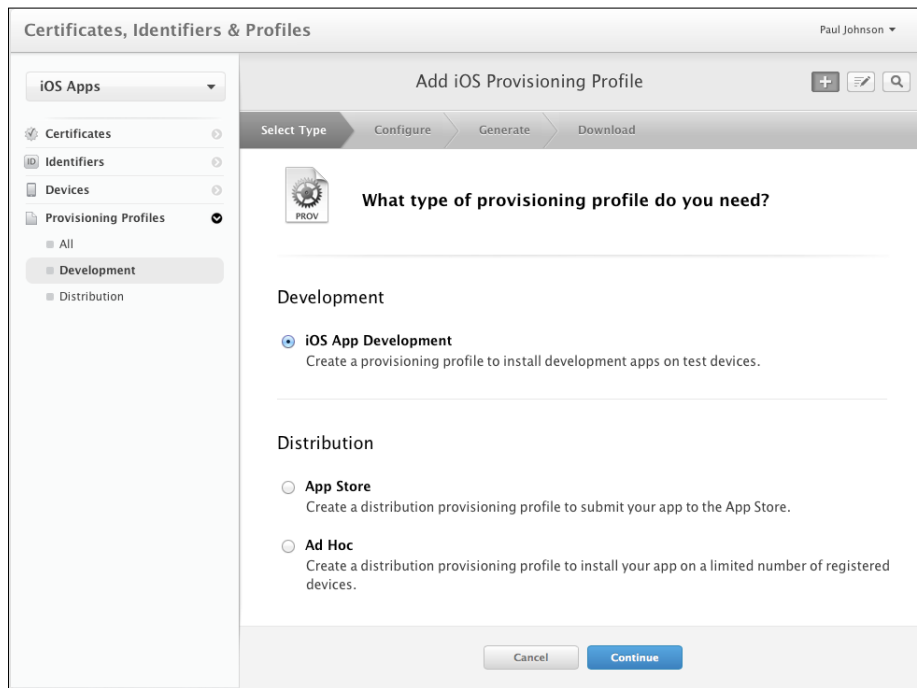
Once you're happy with the values (unless you need the likes of push notifications or in-app purchases, you can just create the ID), you will need to confirm the details. Once they are confirmed, you will be presented with a page detailing all of your apps in development.



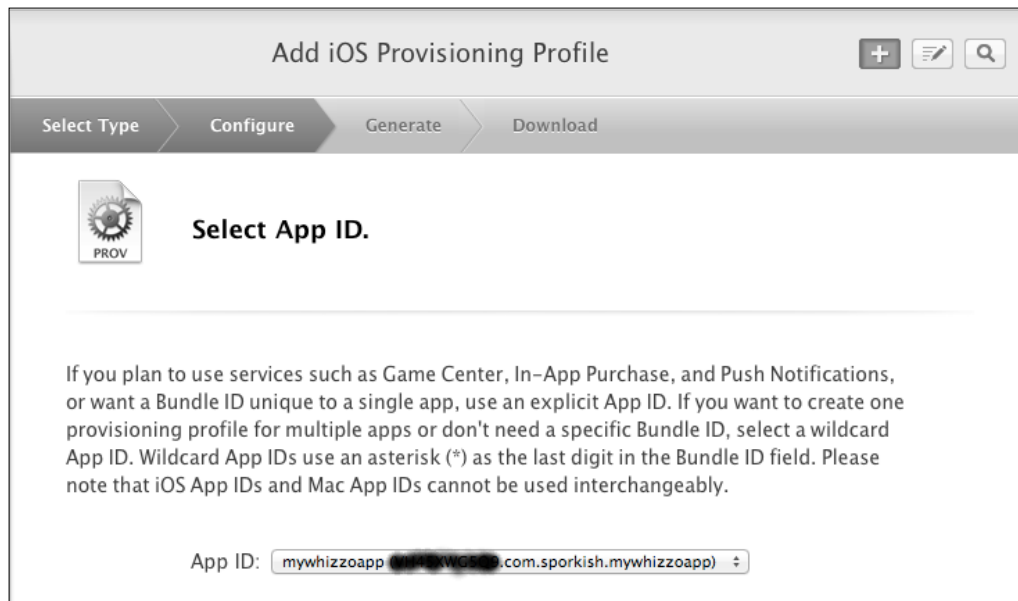
Creating the developer profile

Once you have created the app ID, you need to create a developer profile for it.

1. On the left-side menu, click on **Provisioning Profiles** and then on **Development**. You will get the following screen:



2. Under **Distribution**, you need to select **Ad Hoc**. Click on **Continue**. The next couple of screens are simple to navigate.



3. From the dropdown, select your app. Once it's done, click on **Continue**.
4. Next, you will be presented with a screen asking you to select the certificate you wish to sign the apps with. The certificate is unique to you. If you don't have one, you can now create a new one quickly at this point.

Creating your certificate

Start Xcode and, under the **Window** option, select **Organizer**. Next, select the **Devices** option. You will see a menu on the left. Scroll down to **LIBRARY** and select **Provision Profiles**. You'll need to enter your Apple ID and password. Once done, click on the refresh icon. Xcode will inform you if you don't have a developer profile and give you the option to submit an application for creating one. Click the button with that option in. As soon as you submit the request, the certificate will be added to your keychain. Save the key somewhere safe and give it a username and password. That's it!

You then need to download and install the certificate. On the left-side menu, click on **Certificates**. You will be given an option to load a `.cert` file—this is the file you have just created. Upload the file and once you are done, you will see the following screenshot. Download the file, double-click on it to install, and the job is done. Your certificate is valid as long as you pay money to Apple, and it's all you need for development and distribution.



Back to registering your app

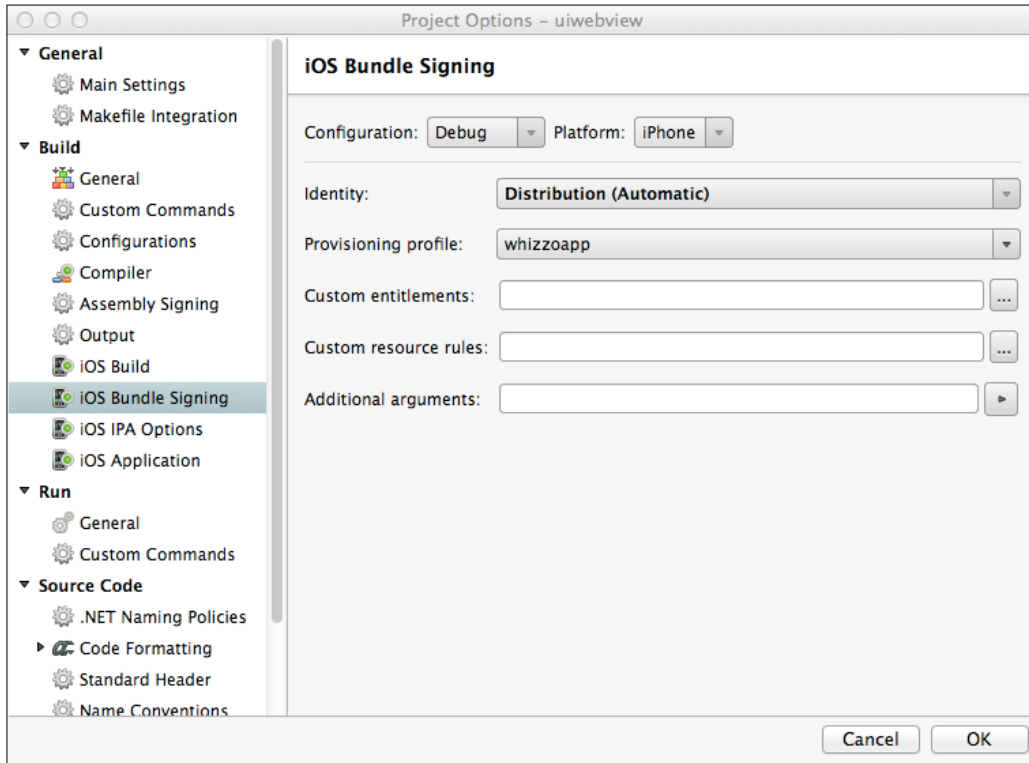
Once you have selected your app, you can then select the devices it needs to go onto. If you have no devices, the page can be skipped and you can download the certificate. Once downloaded, double-click on it, and the profile is installed.

Enabling TestFlight within Xamarin Studio

Xamarin Studio supports `TestFlight`, out of the box. To enable a build using Xamarin Studio, you need to set up the project correctly.

Select your application under **Project**. Next, it needs to be enabled for **AdHoc/Enterprise distribution**. You will find this in the IPA section (check the tick box to enable it).

Finally (for now), select the correct provisioning; here, it is **Distribution (Automatic)** and the name of the app certificate:

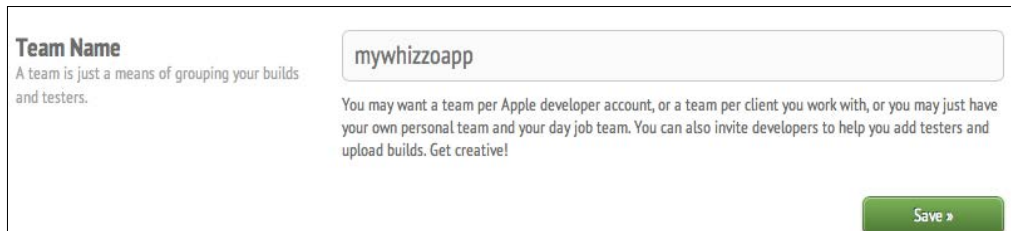


Registering on TestFlight

Signing up on TestFlight is very easy. Create an account and they will provide you with an app and team ID. Don't worry about remembering these since Xamarin Studio will pick up the values when you build out to TestFlight.

Inviting and registering devices

Once registered, you will be invited to create a team. This is essential so you can invite people to test your app. Click on the **Create a new Team** button, type in the name you want to give that area, and click on **Save**.



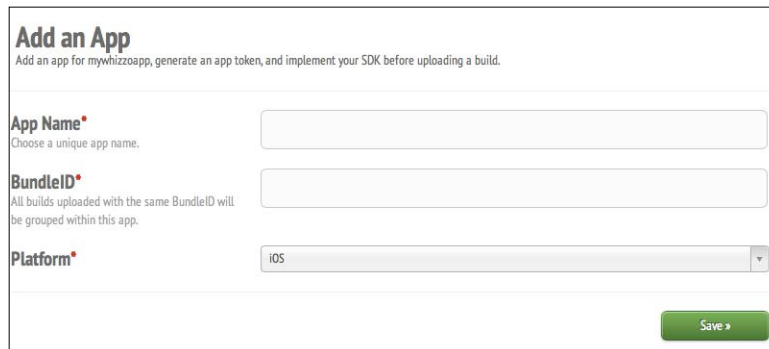
Team Name
A team is just a means of grouping your builds and testers.

You may want a team per Apple developer account, or a team per client you work with, or you may just have your own personal team and your day job team. You can also invite developers to help you add testers and upload builds. Get creative!

Save »

Next, you will be invited to upload a build. This is rather like asking for the car before you've taken lessons. Uploading a build here is pointless as no one will be able to install the app! For this to happen, the testers must have their devices registered on your developer provisioning profile, which only happens once you've invited them.

Before that, though, you have to create an app. Click on the **Apps** link at the top of the page and then on **Create an app**. You will be presented with the following screen:



Add an App
Add an app for mywhizzoapp, generate an app token, and implement your SDK before uploading a build.

App Name*
Choose a unique app name.

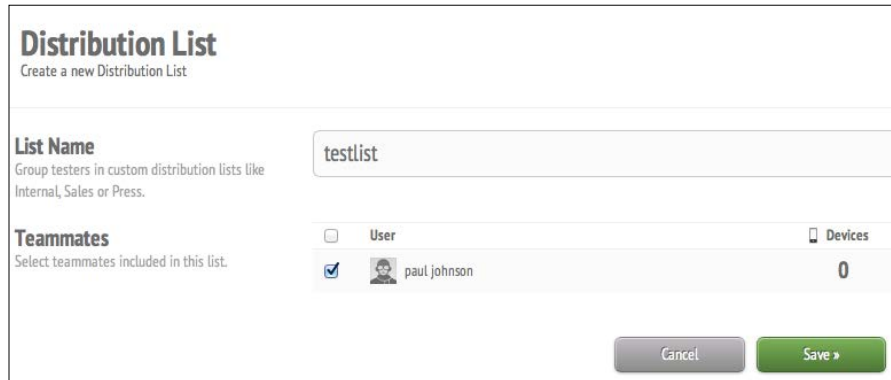
BundleID*
All builds uploaded with the same BundleID will be grouped within this app.

Platform*
iOS

Save »

App Name is the common name for the app (for example, mywhizzoapp). For sanity, keep **BundleID** the same as you have used on the adhoc provisioning profile (com.sporkish.mywhizzoapp). You will need to add this to the application profile in Xamarin Studio (it's under **Bundle**). Once created and saved, you can start inviting people.

Inviting and registering someone is very simple. On the TestFlight website, click on the **People** option at the top. Next, create a new distribution list. These lists are very useful as it means with one account you can distribute a large number of apps to a large number of people.




Distribution List
Create a new Distribution List

List Name
Group testers in custom distribution lists like Internal, Sales or Press.

testlist

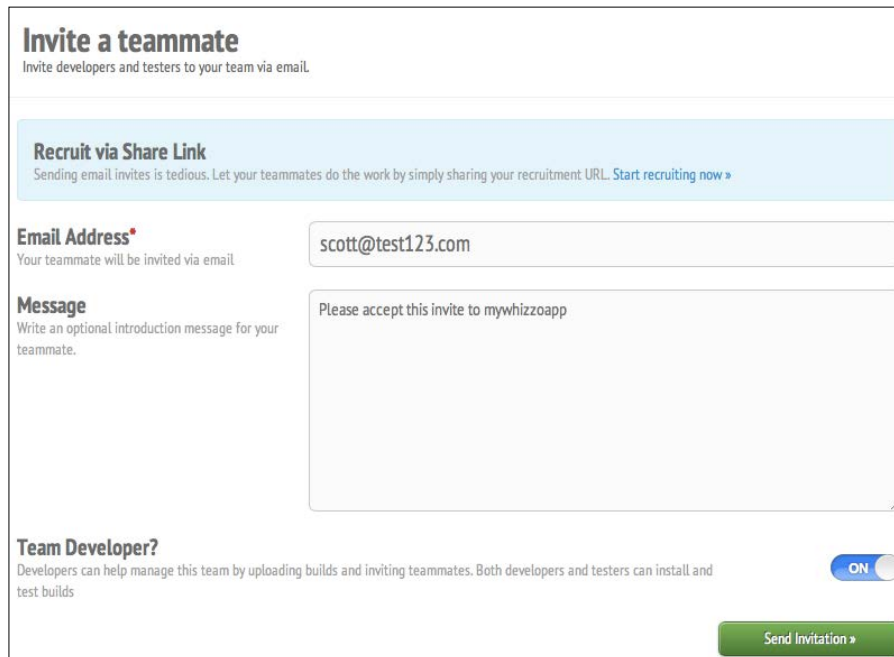
Teammates
Select teammates included in this list.

<input type="checkbox"/>	User	Devices
<input checked="" type="checkbox"/>	 paul johnson	0

Cancel Save »

Give the list a name and, as there is only one person available (that is, you), select your profile, and click on **Save**.

Next, you need to invite people. On the left-side menu, click on **Invitations**. As there are no testers invited, you will be prompted to invite someone.



Invite a teammate
Invite developers and testers to your team via email.

Recruit via Share Link
Sending email invites is tedious. Let your teammates do the work by simply sharing your recruitment URL. [Start recruiting now »](#)

Email Address*
Your teammate will be invited via email.

scott@test123.com

Message
Write an optional introduction message for your teammate.

Please accept this invite to mywhizzoapp

Team Developer?
Developers can help manage this team by uploading builds and inviting teammates. Both developers and testers can install and test builds

ON

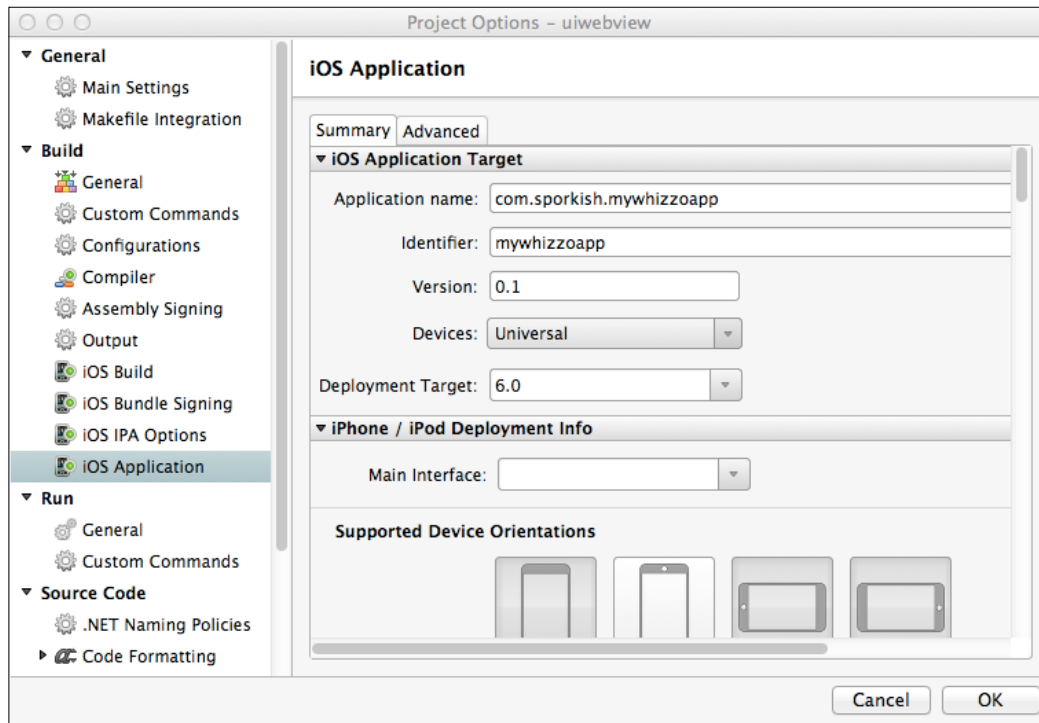
Send Invitation »

If the person you're inviting is also a developer in your team, turn **ON** the **Team Developer** option; otherwise, leave that **OFF**. Once you click on the **Send Invitation** button, an e-mail will be sent to the person, who will in turn, accept the invitation and register the device. You will receive an e-mail once this has been done.

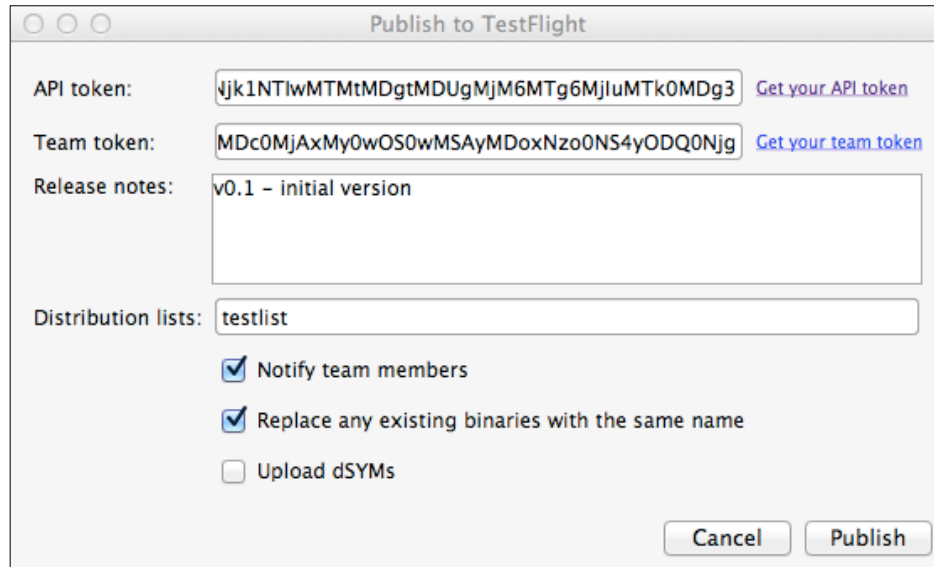
Copy the device `UUID` (or if it has been sent by e-mail, you can upload the file sent containing the device `UUID` code), and go to the Apple developer website. Log in and select **Devices**. Here you can add one or more devices. Once added, go back to the **Developer Provisioning** option and enable that device. You will need to re-download the provisioning file and open it again with Xcode, and then rebuild the app with the updated profile. Failing to do this will result in the user not being able to install the app.

Building to TestFlight

Once you have made sure that the **BundleID** in your project is the same as it is for the **BundleID** of `TestFlight`, things become very easy.



After you have completed entering the details (similar to the preceding screenshot), click on **OK** and set the build to **Ad Hoc**. Select **Project** and then the `TestFlight` option.



Initially, this window will be blank. Click on **Get your API token** and then **Get your team token**, which will launch your default web browser. Copy and paste these values into the **API token** and **Team token** fields on the **Publish to TestFlight** screen (as seen in preceding screenshot). Next, provide some details of what changes are in the file, and finally select the distribution lists you've set up. Ensure that the top two tick boxes are ticked (it helps to let people know when something has been uploaded).

When you're done, click on **Publish** and, assuming there are no build issues, the app will be sent to `TestFlight` and the testers on the notified list.

After a few weeks of testing, you should be ready to submit the app to Apple for distribution. This is the final step to getting your app approved, but there are some hurdles you need to be aware of.

Releasing your app

Testing is complete and you've ironed out a majority of the bugs. There may still be some in there, but for now the app is behaving and you're ready to release it to the world.

App checklist

Before you submit your app to Apple for release through their App Store, you must have the following in place:

- The correct number and size of icons (see below for the sizes)
- The app needs to be correctly signed
- Support for iPhone 5 must be included (this includes the widgets being correctly placed on the screen for the new larger visible area)
- You've not used any private API or forbidden API calls (Apple does not want you using libraries that it does not control and will throw out apps that circumvent hardware or software controls, or interfere with the operation of other software)

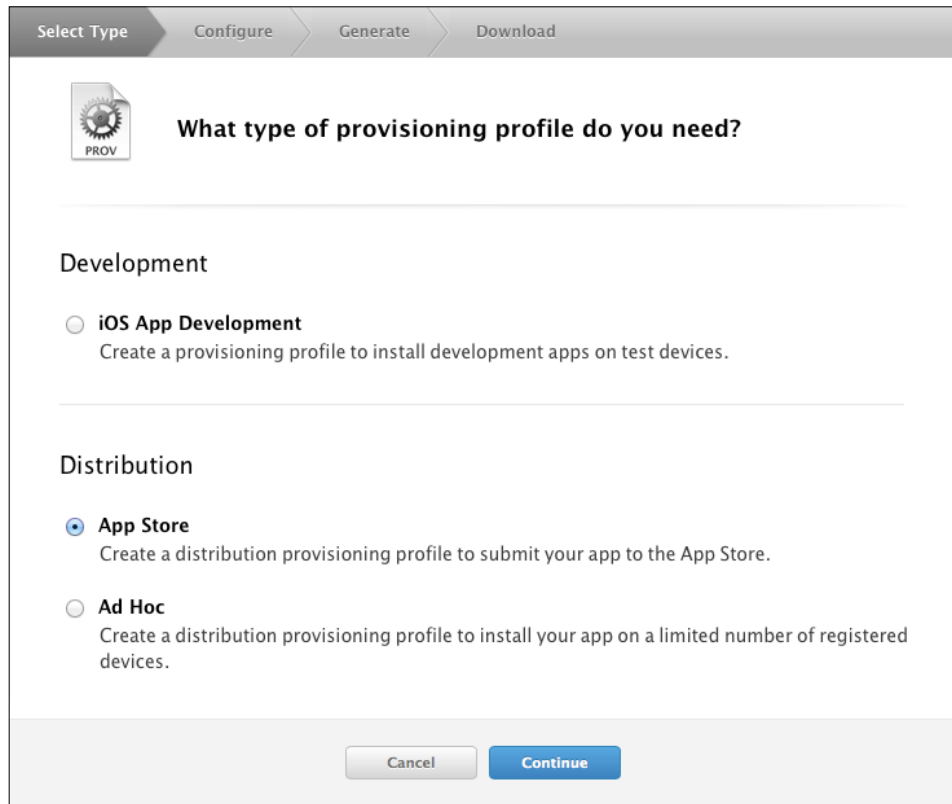
Icon sizes

All of the following icons *must* be present with their correct sizes prior to submitting your app to Apple. The size of the icons (in pixels) are as follows:

iPhone				iPad	
App icons	Normal	Retina	iPad Compatibility	Normal	Retina
	57 x 57	114 x 114	72 x 72	72 x 72	144 x 144
Launch images	Normal	Retina (3.5")	Retina (4")	Normal	Retina
	320 x 480	640 x 960	640 x 1136	768 x 1024	1536 x 2048
iTunes artwork	Normal		Retina		
	512 x 512		1024 x 1024		

Preparing to package

With the icons in place, you next need an App Store provisioning profile. This is set up in a way similar to a **Developer** or **Ad Hoc** provisioning file:



The process for creating the **App Store** profile is similar to the **Ad Hoc** profile generation except that you're not asked which devices you want the test software to be installed on. Download the provisioning file and install it as you did for the **Ad Hoc** profile. You will need to supply a name for the profile, though.

Packaging your app

Once you have created and installed the provisioning profile, you next need to package the app for distribution. This is not as straightforward as it may seem.

You have already created a development certificate; you now need to repeat the process to generate a distribution certificate. The process for doing this is exactly the same as that for a development certificate. Once you have done this, install the certificate in your keychain.

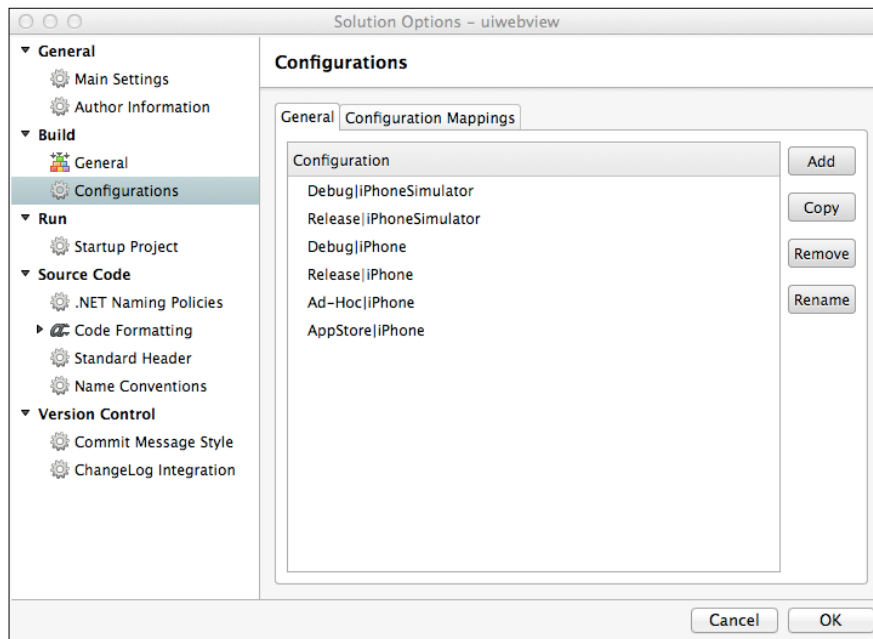
Production

- App Store and Ad Hoc**
Sign your iOS app for submission to the App Store or for Ad Hoc distribution.
- Apple Push Notification service SSL (Production)**
Establish connectivity between your notification server and the Apple Push Notification service production environment. A separate certificate is required for each app you distribute.
- Pass Type ID Certificate**
Sign and send updates to passes in Passbook.
- Website Push ID Certificate**
Sign and send updates for Websites.

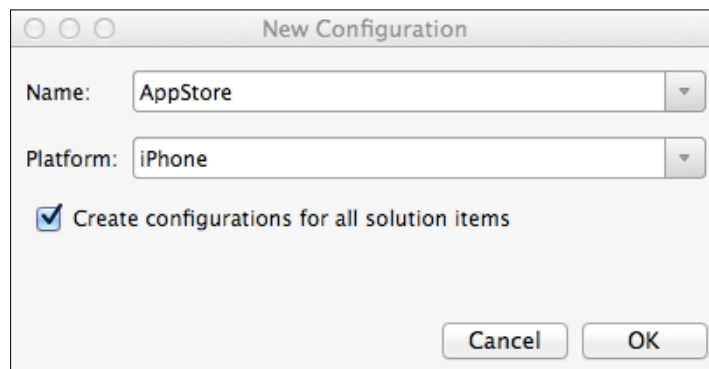
Once the certificate is installed, the next stage takes place in Xamarin Studio.

Creating the build configuration

This is the profile you need for creating your iOS application. Load the project that you wish to submit. Navigate to **Project | Solution Profile**. On the left-hand side, click on the **Configurations** option, shown as follows:



To add the profile (assuming it's not there), click on **Add**. You will be presented with a **New Configuration** box. You will need to fill this in as shown. Once you've done this, click on **OK** and then click on **OK** again to remove the **Solution Options** window.



If the app you've created is universal (in other words, for iPhone and iPad), the platform should be universal. Failing that, for an iPad app, it will be just iPad for the platform.

Next, the application needs to be signed. Navigate to **Project | mywhizzoapp** (or whatever your app is called). Under the **Build** option, select **iOS build** and choose your AppStore configuration. After this, select the iOS bundle signing option. For the Identity drop-down, set this to Distribution (automatic) and the provisioning profile to the one you created to distribute the app (not the one created for distribution on TestFlight).

Once this has been done, click on **OK** to remove the window and set the build type for the app to **AppStore**.



For safety, click on **Build | Clean All** and then **Build | Build all**. If everything works, we're in the App Store.

The App Store Submission Process

The first step is to create a record with **iTunes Connect** (<https://itunesconnect.apple.com>). This is a simple process. The description has to reflect what the app can do (for example, you cannot submit a messenger application that is actually a game).

Assuming that the record has been created correctly, click on the **Ready To Upload** binary button. You will be presented with a number of questions, and you can progress by answering them correctly. The last screen will tell you that the App Store is ready to upload and you can use the uploader facility. But *don't*. There is a simpler method via Xamarin Studio and Xcode, which has fewer issues. The first part is to create an archive.

Creating an archive

Navigate to **Build | Archive**. This will build the app and create an archive from it that is correctly provisioned for distribution. Once the archive has been built, an **Archive** tab will appear in the code designer. Here you can look at any of the archives that you have created.

Submission via Xcode

Start Xcode and navigate to **Window | Organiser**. Click on the **Archives** tab. You will be presented with a list of all your apps for distribution. Select the one you wish to submit and you will see two options: **Validate** and **Submit**. Click on **Validate**. If there are no issues with the app, proceed to **Submit**. Typically, validation fails if there isn't a correct provisioning certificate applied to the app or the certificate can't be found. Fix the issue, rearchive, and validate again.

The submission wizard

To guide you through the process of submission, Apple has included a very simple-to-use submission wizard. You must ensure that you have a network connection for this stage and that the app is ready for upload.

Click on **Submit**. Xcode communicates with **iTunes Connect** and retrieves a list of apps you have submitted for upload. Select the one you want to submit. Xcode will then upload the archive for you. Once uploaded, if you go back to **iTunes Connect**, you will see that the app is now waiting for approval. (Don't worry as this is largely an automated procedure.) Once approved, you're home and dry and your app is in the App Store.

Summary

Testing and distribution of iOS applications may seem a waste of time, but at the end of the day, controlling who is testing your app allows you to pick and choose who you want to test the code. Moreover, it doesn't allow for versions to be leaked and in turn giving you a potentially bad name. Yes, it may seem very control-freaky, but that's the game Apple wants you to play, and if you want to distribute your apps with them, that's the game you have to play.

Index

Symbols

.caf (Core Audio File) 157
.plist file
 about 162
 reading from 163
 writing to 163

A

ABAddressBook 75
ABNewPersonViewController 75
ABPeoplePickerNavigationController 75
ABPersonViewController 76
ABUnknownPersonViewController 76
Acceleration event 90
AcceptEvent event 80
AccessoryTapped event 83
action 17
ActionFinished event 96
ActivityElement 23
Activity Indicator View 38
AdBanner View 38
AdBannerView 96 45
Ad classes
 AdBannerView 96
 AdInterstitialAd 96
 IGameWindow 96
 iPhoneOSGameView 97
 OpenTK 96
AddressResolved event 85
AdInterstitialAd 96
AdLoaded event 45, 96
AdUnloaded event 96
AdvertisingStarted event 79
Ahead Of Time (AOT) 135

AllEditingEvents event 74
AllEvents event 74
AllTouchEvents event 74
Android
 comparing, to iOS Ui controls 67, 68
Android SDK Manager 10
Android Software Developers Kit. *See* SDK
animation 123
AnimationEnded event 77
AnimationStarted event 77
AnimationWillEnd event 95
AnimationWillStart event 95
app
 checklist 180
 packaging 182
 registering 171-174
 releasing 179
AppDelegate class 114, 131
App Store Submission Process
 about 184
 archive, creating 184
 submission wizard 185
 via Xcode 185
Asynchronous code
 using 121
asynchronous event handling
 versus synchronous event handling 71
asynchronous walk 71
async method 121
audio
 NSDictionary, setting up 159
AudioConverter 76, 77
audio file
 playing 158
 recording 159
AudioRouteChanged event 76

AudioSession 76
AudioUnit 77
AUGraph 77
AuthorizationChanged event 80
AVAudioPlayer 74
AVAudioSession 74
AVRecordClass 74

B

backBarButtonItem 56, 57
background threading
 using, within app 116
BackgroundWorker DoWork thread 118
BackgroundWorker thread 116
BadgeElement 23, 82
BarButtonItem 66
BaseBooleanImageElement 23, 82
BeginAnimations 125
BeginInterruption event 74
bitmaps
 bindings, underpinning 125
 code, analysis 125, 126
 handling 123
 image, rotating 124, 125
 image, scaling 123, 124
Bonjour 7
BookmarkButtonClicked event 93
BooleanElement 23
BooleanImageElement 23
BoolElement 23, 82
build configuration
 creating 182-184
built-in system 162, 163

C

CAAnimation 77
CalendarChooser class 83
callback 120
callLoginService 122
callNewMethod 127
CalloutAccessoryControlTapped event 88
camera
 about 139
 accessing, UIImagePickerController
 used 140

 Xamarin.Mobile component 140
CanAcceptBytesEvent event 80
CancelButtonClicked event 93
Cancelled event 75, 83, 91
cardex system 130
CategoryChanged event 74
CBCentralManager 78
CBPeripheral 78
CBPeripheralManager 79
certificate
 creating 173, 174
CFSocket 80
CFStream 80
CGColor 31
ChangedDragState event 88
Changed event 83, 94
CharacteristicSubscribed event 79
CharacteristicUnsubscribed event 79
CheckboxElement 23
CIColor 31
Clicked event 91
CLLocationManager 80, 81
Closed event 97
ClosedEvent event 80
Collection Reusable View 38
Collection View 38
CommitAnimations 126
Completed event 83
ConnectedPeripheral event 78
ConnectEvent event 80
ConnectionAutomatic flag 152
ConnectionFailed event 87
ConnectionOnDemand flag 152
ConnectionOnTraffic flag 152
ConnectionRequest event 87
ConnectionRequired flag 152
connector 17
controls
 list 66, 67
 selection 62
CoreAnimation namespace 123, 124
CoreBluetoothCentralManager 78
CoreGraphics namespace 123
Core Location
 about 141, 143
 delegate, setting 143, 144

- map, adding 147-149
- pin, adding 150
- setting 143
- users, finding 145, 146
- using 143

CoreLocation LocationManager class 80

D

data

- adding, to database 134
- deserializing 164, 165
- manipulating, LINQ used 135
- serializing 164, 165

database

- basics 130
- class 130
- connection, creating 131, 132

data class 167

DataManager class 131

DataReceived event 87

DateElement 23

DatePicker 66

DateSelected event 82

DateTimeElement 23, 82

DateTime Picker View 82

DBClauseSyncOff constant 132

DBClauseVacuum constant 132

deadlocking

- about 108
- avoiding, for synchronized accessors 109

DecelerationEnded event 92-94

DecelerationStarted event 92-94

DecodeError event 74

DeferredUpdatesFinished event 80

delegate event

- about 69, 70
- attaching, to multiple controls 70

demoTable variable 130

deserializing data 164, 165

DesiredAccuracy 142

DetailDisclosureButton 51

DetailTextLabel 51

developer profile

- creating 172, 173

DialogViewController 82

DidAddAnnotationViews event 88

DidAddOverlayViews event 88

DidBeginCustomizingItems event 93

DidCancel event 89

DidChangeUserTrackingModel event 88

DidDeselectAnnotationView event 88

DidDismiss event 89, 92

DidEndCustomizingItems event 93

DidFailToLocateUser event 88

DidFailWithError event 87

DidFindMatch event 87

DidFindPlayers event 87

DidFinishAnimating event 91

DidFinish event 86

DidFinishJob event 92

DidPresentPrinterOptions event 92

DidSelectAnnotationView event 88

DidShowViewController event 91

DidStartMonitoringForRegion event 80

DidStopLocatingUser event 88

DidUpdateUserLocation event 88

DidZoom event 92-94

Disclosure 51

Disconnected event 84

DisconnectedPeripheral event 78

DiscoverCharacteristic event 78

DiscoveredDescriptor event 78

DiscoveredIncludedService event 78

DiscoveredPeripheral event 78

DiscoveredService event 78

Dismissed event 91

Disposed event 97

DomainRemoved event 85

drag-and-drop 104, 105

DraggingEnded event 92, 94

DraggingStarted event 92, 94

DrawInRect event 86

Duration 125

E

EAAccessory 84

EaseIn 126

EaseInOut 126

EaseOut 126

EditingChanged event 73

EditingDidBegin event 73

EditingDidEnd event 74

- EditingDidEndOnExit event 74
- EKCalendarChooser 83
- EKEventEditViewController 83
- EKEventViewController 83
- element
 - about 18, 23
 - types 18-20
- EncodedObject event 84
- EncoderCallback event 77
- Ended event 94
- EndInterruption event 74
- EntryElement 24, 83
- EntryElement class 21, 22
- ErrorEvent event 80
- EventArgs e 70
- events
 - handling 69
 - handling, delegates used 69, 70
- ExecuteScalar method 134
- ExternalChange event 75

F

- Failed event 80, 87
- failed, recognizer state 104
- FailedToConnectPeripheral event 78
- FailedToReceive 45
- FailedToReceiveAd event 96
- FinishedCustomizingViewController event 94
- Finished event 83-90
- FinishedPickingImage event 91
- FinishedPickingMedia event 91
- FinishedPickingMedia() method 141
- FinishedPlaying event 74
- Finishing event 84, 85
- Fixed/FlexibleBarButtonItem 67
- FloatElement 24
- FoundDomain event 85
- FoundService event 85
- FromBundle 40
- FromFile 40
- FromImage 40
- FromResource 40
- FromWhiteAlpha 30

G

- garbage collector (GC) 126
- gestures
 - about 99-101
 - adding 100, 101
 - adding, in code 102
 - code 101, 102
 - continuous types 102
 - drag-and-drop, handling 104, 105
 - Selector method 102
 - tapGesture 101
 - types 102
 - UILongPressGestureRecognizer class 100
 - UIPanGestureRecognizer class 99
 - UIPinchGestureRecognizer class 100
 - UIRotationGestureRecognizer class 100
 - UISwipeGestureRecognizer class 99
 - UITapGestureRecognizer class 99
- GetCell method 51, 52
- GKAchievementViewController 86
- GK classes
 - GKAchievementViewController 86
 - GKFriendRequestComposeViewController 86
 - GKGameCenterViewController 86
 - GKLeaderboardViewController 86
 - GKMatch 87
 - GKMatchmakerViewController 87
 - GKSession 87
- GKFriendRequestComposeViewController 86
- GKGameCenterViewController 86
- GKLeaderboardViewController 86
- GKMatch 87
- GKMatchmakerViewController 87
- GKSession 87
- GlassButton 82
- GLKit 59
- GLKit View 38
- GLKView 86

H

- handler class
 - about 165
 - accessor, setting up 167

- handler code 37
- HandleScanResult method 72
- HasBytesAvailableEvent event 80
- HtmlElement 24

I

- IGameWindow 96
- image
 - adding 32
 - bitmap image, handling 123
 - rotating 124, 125, 128
 - scaling 123, 124
- ImageElement 24
- ImageStringElement 24
- ImageView 38
- InputAccessoryView method 21
- InputAudioQueue 77
- InputAvailabilityChanged event 74
- InputChannelsChanged event 74
- InputCompleted event 77
- InputData event 76
- Internet
 - accessing 152-155
- Interrupted event 76
- InterventionRequired flag 152
- InvalidatedService event 78
- InvokeOnMainThread method 39
- IOError event 81
- iOS Layout
 - about 36
 - canvas model 36
 - issues, avoiding 37
- IOStream 80
- iOS UI controls
 - and Android, comparing 67, 68
- iPhoneOSGameView 97
- IsDirect flag 152
- IsLocalDevice flag 152
- IsSourceTypeAvailable(myCamera)
 - command 155
- IsWWAN flag 152
- ItemSelected event 93
- ItemsPicked event 89
- iTunes Connect
 - URL 184

J

- JsonElement 24

K

- keyboard

- toolbar, adding 21
 - type, changing 20

L

- Label 66

- leftBarButtonItem 56

- LINQ

- about 136
 - SELECT 137
 - SELECT, using 137, 138
 - SQL, replacing with 138
 - used, for manipulating data 135
 - WHERE 137

- ListButtonClicked event 93

- Liverpool 50

- LoadError event 95

- Load event 96, 97

- LoadFinished event 95

- LoadFromData 40

- LoadingMapFailed event 88

- LoadMoreElement 24, 82

- LoadStarted event 95, 121

- LocationsUpdated event 80

- LocationUpdatesPaused event 80

- LocationUpdatesResumed event 80

- locks 112, 113

M

- map

- adding 147, 148

- MapLoaded event 88

- Map View 38

- MapView 44

- mapViewer 147

- Master-Detail, project type 36

- memory

- freeing, after use 126, 127
 - image, rotating 128

MessageElement 24, 82
MessageReceived event 81
MFMailComposeViewController 89
MFMessageComposeViewController 89
MidiClient 81
MidiEndpoint 81
MidiPort 81
MKMapView 88
MKReverseGeocoder class 145
Model View Controller (MVC) 47
MonitoringFailed event 81
Monotouch.CoreGraphics namespace 64
MonoTouch.Dialog. *See* **MT.D**
MonoTouch.MapKit namespace 145
MonoTouch.MediaPlayer namespace 155
MPMediaPickerController 89
MT.D
 about 18-20, 81
 keyboard type, changing 20
 Pickers, creating 25-29
 ResignFirstResponder, using 21
 ShouldReturn, using 20
 toolbar, adding to keyboard 21
 types, supported 23
MT.D views
 element 18
 root 18
 section 18
MultilineElement 24
multimedia
 about 155
 audio, recording 158
 audio system 157
 video, playing 155
 video, recording 156
multiple view controllers
 used, for implementing view 45, 46

N

NavigationBar 66
NavigationController 53-56
NavigationItem 66
NewPersonComplete event 75
NewPersonViewController 75
NextStep (NS) 84
NotSearched event 85

NSAutoreleasePool class 39
NSCache 84
NS classes
 NSCache 84
 NSKeyedArchiver 84
 NSKeyedUnarchiver 85
 NSNetService 85
 NSNetServiceBrowser 85
 NSStream 86
NSKeyedArchiver 84
NSKeyedUnarchiver 85
NSNetService 85
NSNetServiceBrowser 85
NSNotificationCenter class 163
NSObject class 163
NSStream 86
NSUserDefaults class 163

O

ObjectAdded event 81
ObjectRemoved event 81
OnCustomizingViewController event 94
OnEditingStarted event 93
OnEditingStopped event 93
OnEndCustomizingViewController
 event 94
OnEvent event 86
OnSelection event 82
OpenCompletedEvent event 80
OpenGL for Embedded Systems
 (OpenGL ES) 59
OpenGL, project type 36
OpenTK 96
outlet 17
outlet collection 17
OutputAudioQueue 77
OutputChannelsChanged event 75
OutputCompleted event 77
OwnerDrawnElement 24

P

package
 app 182
 build configuration, creating 182-184
 preparing to 181

- PageControl** 66 59
- PeerChanged** event 87
- PerformAction** event 75
- PerformDefaultAction** event 76
- PersonCreated** event 76
- phone**
 - storage 150
- phone call**
 - making 150
- Pickers**
 - creating, on MT.D 25-29
- Picker View** 38
- pin**
 - adding 150
- PKAddPassesViewController** 89
- playback**
 - about 157
 - audio file, playing 158
 - long (and compressed) files 157
 - power levels, setting 158
 - short files 157
 - volume, altering 158
- possible, recognizer state** 104
- power levels**
 - setting 158
- Presented** event 91
- ProgressChanged** event 116
- Progress View** 38
- project, types**
 - about 36
 - Master-Detail 36
 - OpenGL 36
 - Single View 36
 - Tabbed 36
 - URL 36
- PropertyChanged** event 81
- provisioning** 170
- provisioning profile** 169, 170
- Published** event 85
- PublishFeature** event 85

Q

- QLPreviewController** 89
- QueueUserWorkItem** method 120

R

- RadioElement** 24
- Reachable** flag 152
- read-only table**
 - creating 49-51
- ReadRequestReceived** event 79
- ReadyToUpdateSubscribers** event 79
- ReceivedAcceptFromHostedPlayer** event 87
- ReceiveData** event 87
- ReceivedResponse** event 90
- record**
 - setting up 159
- RefreshRequested** event 82
- RegionChanged** event 88
- RegionEntered** event 81
- RegionLeft** event 81
- RegionWillChange** event 88
- RenderCallback** event 77
- RenderedFrame** event 96, 97
- RepeatAnimationCurve** 126
- ReplacingObject** event 84, 85
- RequestFailed** event 90
- RequestFinished** event 90
- ResignFirstResponder**
 - using 21
- ResolveFailed** event 85
- Resumed** event 76
- RetrievedConnectedPeripheral** event 78
- RetrievedPeripherals** event 78
- retString** variable 137
- ReturnKeyType** 20
- rightBarButtonItem** 56
- root** 18
- RootElement** 24
- RssiUpdated** event 78
- RunWorkerCompleted** event 116

S

- SampleRateChanged** event 75
- ScaleFactor** 123
- screen**
 - origins 18
 - sizes 18
- ScrollAnimationEnded** event 92-94

- Scrolled event 92-94
- ScrolledToTop event 92, 95
- ScrollEnabled 149
- Scroll View 38
- ScrollView 94
- SDK 10
- SearchBar 66
- SearchButtonClicked event 93
- SearchRemoved event 85
- SearchStarted event 85
- SearchStopped event 85
- SearchTextChanged event 82
- section 18
- SectionIndexTitles 53
- SegmentedControl 66
- SELECT
 - using, in LINQ 137
- SelectedScopeButtonIndexChanged event 93
- SelectionChanged event 83, 95
- Selector method 102
- SelectPerson event 75
- serializing data 164, 165
- SerialPortOwnerChanged event 81
- ServiceAdded event 79
- setImage method 32
- Settings.bundle 162
- settings file
 - data class 167
 - handler class 165
 - setting up 165
- settings system
 - about 164
 - file, setting up 165
- SetupChanged event 81
- ShouldReturn
 - using 20
- ShouldReturn event 83
- ShowUserLocation 149
- Single View, project type 36
- SK classes
 - SKProductsRequest 90
 - SKRequest 90
 - SKStoreProductViewController 90
 - SKProductsRequest 90
 - SKRequest 90
- SKStoreProductViewController 90
- Sleep() call 109
- Slider 66
- speed
 - calculating 143
- SQL
 - replacing, with LINQ 138
- SQLite
 - about 129
 - installing 129
- SQLite helper class
 - about 132
 - data, adding to database 134
 - methods 133, 134
- Started event 94, 95
- StartListening 142
- StateChanged event 87
- StateUpdated event 79
- StopListening 142
- Stopped event 85
- StringElement 25, 82
- StyledMultilineElement 25
- StyledStringElement 25, 83
- synchronous event handling
 - versus asynchronous event handling 71
- synchronous walk 71
- system
 - built-in settings 164
- System.ComponentModel namespace 116
- SystemSound method 157
- System.Threading.Tasks namespace 120

T

- TabBarItem 67
- TabBars
 - about 57, 58, 67
 - handling, in code 58
- Tabbed, project type 36
- TableView
 - about 38
 - indexes 53
- TableViewCell 51
- TableView method 47
- tapGesture 101
- Tapped event 82

- tasks**
 - and EventHandlers 121
 - using on threads, issues faced 120
- Task<T> parameter 122**
- testappViewController.cs file 15**
- testappViewController.designer.cs file 15**
- testappViewController_iPad.xib file 15**
- testappViewController_iPhone.xib file 15**
- TestFlight**
 - about 170
 - app, checklist 180
 - app, registering 171, 174
 - app, releasing 179
 - building to 178, 179
 - certificate, creating 173, 174
 - developer profile, creating 172, 173
 - devices, inviting 176-178
 - enabling, within Xamarin Studio 174
 - icon, sizes 180
 - provisioning 170
 - registering on 175
- TextChanged event 93**
- TextField 66**
- TextLabel 51**
- text message**
 - receiving 151
 - sending 151
- Text View 38**
- threading**
 - about 107, 115
 - background threading, using within app 116
 - BackgroundWorker thread 116-118
 - deadlocking 108
 - locks, using 112, 113
 - new thread, starting from main
 - UI thread 109-112
 - System.Threading.Tasks, using 120
 - ThreadPool.QueueUserWorkItem 119, 120
 - UI thread 108
- ThreadPool.QueueUserWorkItem 119, 120**
- ThruConnectionsChanged event 81**
- TimeElement 25**
- toolbar**
 - about 66
 - adding, to keyboard 21
- TouchCancel event 73**
- TouchDown event 73**
- TouchDownRepeat event 73**
- TouchDragEnter event 73**
- TouchDragExit event 73**
- TouchDragInside event 73**
- TouchDragOutside event 73**
- TouchUpInside event 70-73**
- TouchUpOutside event 73**
- TransientConnection flag 152**
- translation, recognizer state 104**

U

- UIAccelerometer 90**
- UIActionSheet 91**
- UIActivityIndicatorView class 39**
- UIAlertView 91, 123**
- UIButton 31, 32, 63, 64**
- UIBarButtonItem 91**
- UIButton class 62**
- UIClasses**
 - ScrollView 94
 - UIAccelerometer 90
 - UIActionSheet 91
 - UIAlertView 91
 - UIBarButtonItem 91
 - UIImagePickerController 91
 - UIPageViewController 91
 - UIPopoverController 92
 - UIPrintInteractionController 92
 - UIScrollView 92
 - UISearchBar 93
 - UISplitViewController 93
 - UITabBar 93
 - UITabBarController 94
 - UITextField 94
 - UITextView 94
 - UIView 95
 - UIWebView 95
- UICollectionView**
 - about 41
 - cell reuse 42
 - cells 42
 - data source 42
 - decoration view 42

- supplementary views 42
- UICollectionView class** 42
- UIColor** 30, 31
- UI Controls**
 - about 62
 - selection 62
- UIControlStates** 33
- UIGestureRecognizerState values**
 - failed state 104
 - possible state 104
 - translation state 104
 - velocity state 104
- UIImagePickerController**
 - about 91
 - used, for accessing camera 140
- UIImageView**
 - about 40, 67, 127
 - FromBundle 40
 - FromFile 40
 - FromImage 40
 - FromResource 40
 - LoadFromData 40
- UILabel** 29
- UILabel class** 64
- UILongPressGestureRecognizer class** 100
- UIPageControl method** 59
- UIPageViewController** 91
- UIPanGestureRecognizer class** 99
- UIPinchGestureRecognizer class** 100
- UIPopoverController** 92
- UIPrintInteractionController** 92
- UIProgressView class** 39
- UIRotationGestureRecognizer class** 100
- UIScrollView** 92 44
- UISearchBar** 93
- UISplitViewController** 93
- UIStepper** 65
- UISwipeGestureRecognizer class** 99
- UITabBar** 93
- UITabBarController** 94
- UITableView**
 - navigation with 53
 - RootView, returning to 57
 - used, for navigation 56
 - within code 53
 - with Xcode 54-56

- UITableViewCell**
 - about 51
 - cells, reusable within table 52
 - indexes, on TableView 53
 - rows 52
 - sections 52
- UITableViewCell method** 29
- UITableViewCellStyle**
 - Subtitle 51
 - Value1 51
 - Value2 51
- UITableView method** 29
- UITapGestureRecognizer class** 99
- UITextField method** 22
- UITextView** 22, 94
- UI thread**
 - about 108
 - new thread, starting from 110, 111
- UIView** 95
- UIViewElement** 25
- UIWebView** 42, 43, 95
- Unknown Person View Controllers** 76
- Unload event** 96, 97
- UpdatedCharacteristicValues event** 78
- UpdatedHeading event** 81
- UpdatedLocation event** 81
- UpdatedName event** 79
- UpdatedNotificationState event** 79
- UpdatedState event** 78
- UpdatedTxtRecordData event** 85
- UpdatedValue event** 79
- UpdateFrame event** 96, 97
- UserInteractionEnabled** 149
- user interface**
 - about 13
 - creating, Xcode used 13-17

V

- ValueChanged event** 73, 82
- velocity, recognizer state** 104
- video**
 - recording 156
 - saving 157
- video, playing**
 - about 155

- external URL 155
- from photo library 156
- internal source 155

view

- about 37
- Activity Indicator View 38
- AdBanner View 38
- Collection Reusable View 38
- Collection View 38
- GLKit View 38
- ImageView 38
- implementing, with multiple view controllers 45, 46
- Map View 38
- Picker View 38
- Progress View 38
- Scroll View 38
- Table View 38
- Text View 38
- Web View 38

ViewAppearing event 82

view controller 37

ViewControllerSelected event 94

ViewDidLoad() method 44

ViewDisappearing event 82

VisibleChanged event 97

Visual Studio

- enabling, to build iOS applications 6, 7
- enabling, to run iOS applications 6
- on Mac 7
- on PC 7-9

volume

- altering 158

W

WasCancelled event 87

Web View 38

WHERE syntax 137

widget

- adding 17
- connecting 17

WillBeginCustomizingItems event 93

WillDismiss event 89, 91

WillDismissPrinterOptions event 92

WillEndCustomizingItems event 93

WillEndDragging event 92, 95

WillEvictObject event 84

WillHideViewController event 93

WillLoad event 96

WillPresent event 91

WillPresentPrinterOptions event 92

WillPresentViewController event 93

WillPublish event 85

WillResolve event 85

WillShowViewController event 91, 93

WillStartJob event 92

WillStartLoadingMap event 88

WillStartLocatingUser event 88

WillTransition event 91

WInDow gadGET 61

Windows 7

- Bonjour service, setting up 9

WindowStateChangeEvent 97

WorkerCompleted event 116

WriteRequestsReceived event 79

WroteCharacteristicValue event 79

WroteDescriptorValue event 79

X

Xamarin.Android

- about 10
- installing 5
- requisites 5
- software, downloading 6
- software, installing 6

Xamarin IDE (Integrated Development Environment) 10

XamarinInstaller.exe file 6

Xamarin.iOS

- installing 5
- requisites 5
- software, downloading 6
- software, installing 6

Xamarin.iOS Visual Studio

- on Mac 7
- on PC 7, 8, 9

Xamarin.Mobile component

- about 139-141
- GPS with 141, 142

Xamarin Studio

- used, for enabling TestFlight 174

Xcode

- about 6
- starting 15
- used, for creating user interface 13-17

Xcode designer

- adding 16

Z

ZoomEnabled 149

ZoomingEnded event 92, 95

ZoomingStarted event 92, 95



Thank you for buying **Xamarin Mobile Application Development for iOS**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



iOS Development using MonoTouch Cookbook

ISBN: 978-1-849691-46-8 Paperback: 384 pages

109 simple but incredibly effective recipes for developing and deploying applications for iOS using C# and .NET

1. Detailed examples covering every aspect of iOS development using MonoTouch and C#/.NET
2. Create fully working MonoTouch projects using step-by-step instructions.
3. Recipes for creating iOS applications meeting Apple's guidelines.



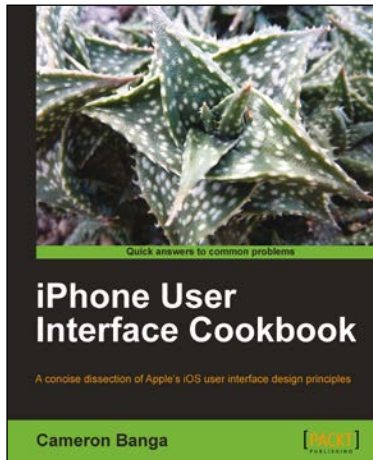
HTML5 iPhone Web Application Development

ISBN: 978-1-849691-02-4 Paperback: 338 pages

An introduction to web-application development for mobile within the iOS Safari browser

1. Simple and complex problems will be covered with examples and resources that backup the approach and technique.
2. Real world solutions that are broken down for multiple target audiences; from beginner developers to technical architects.
3. Learn to build true web applications using the latest industry standards for iOS Safari.

Please check www.PacktPub.com for information on our titles

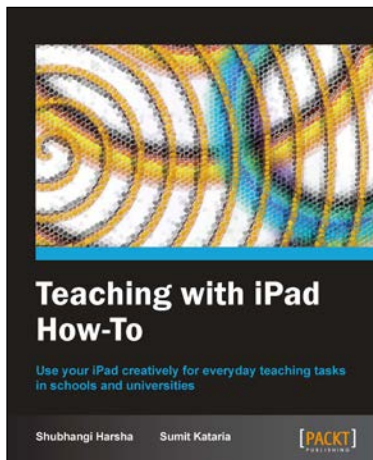


iPhone User Interface Cookbook

ISBN: 978-1-849691-14-7 Paperback: 262 pages

A concise dissection of Apple's iOS user interface design principles

1. Learn how to build an intuitive interface for your future iOS application
2. Avoid app rejection with detailed insight into how to best abide by Apple's interface guidelines
3. Written for designers new to iOS, who may be unfamiliar with Objective-C or coding an interface



Teaching with iPad How-To

ISBN: 978-1-849694-42-1 Paperback: 240 pages

Use your iPad creatively for everyday teaching tasks in schools and universities

1. Learn something new in an Instant!
A short, fast, focused guide delivering immediate results
2. Plan your lessons on iPad and share notes quickly
3. Use exclusive iPad 3D resources for more engaging learning
4. Use your iPad for creating and giving presentations

Please check www.PacktPub.com for information on our titles