



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 Essentials

Learn how to efficiently develop Android and iOS apps for deployment using the Xamarin platform

Mark Reynolds

www.allitebooks.com

[PACKT]
PUBLISHING

Xamarin Essentials

Learn how to efficiently develop Android and iOS apps for deployment using the Xamarin platform

Mark Reynolds



BIRMINGHAM - MUMBAI

Xamarin Essentials

Copyright © 2014 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: December 2014

Production reference: 1221214

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

ISBN 978-1-78355-083-8

www.packtpub.com

Cover image by Abhishek Dhir (abhishekdhirimages@gmail.com)

Credits

Author

Mark Reynolds

Project Coordinator

Mary Alex

Reviewers

Jason Awbrey

Joe Dan Galyean

Paul F. Johnson

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Paul Hindle

Commissioning Editor

Amarabha Banerjee

Indexer

Monica Ajmera Mehta

Acquisition Editor

Richard Harvey

Graphics

Abhinash Sahu

Content Development Editors

Priya Singh

Production Coordinator

Rohit Singh

Conidon Miranda

Technical Editor

Nitee Shetty

Cover Work

Conidon Miranda

Copy Editors

Relin Hedly

Stuti Srivastava

Neha Vyas

About the Author

Mark Reynolds is a software enthusiast who has worked in the industry for nearly 30 years. He began his career with Electronic Data Systems, building and supporting systems for the manufacturing sector. Over the years, Mark has worked with start-ups and Fortune 100 companies across a diverse set of industries. In 1993, he started a consulting practice that focused on delivering training and mentoring services in the areas of software architecture, design, and implementation. With the rise of mobile computing, he has focused his practice on designing, developing, and delivering mobile software solutions.

Mark recently published his first book, *Xamarin Mobile Application Development for Android*, Packt Publishing. His private consulting practice is based in Allen, TX, USA, where he resides with his wife and son.

I would like to thank my mother, Charlene Reynolds, who I lost this year. She was a great mother, wife, sister, cousin, aunt, and friend, and a great inspiration to everyone she came in contact with. We'll miss her, but we know in her new home; she has overcome the illness she battled here, and we rejoice in that.

About the Reviewers

Jason Awbrey is a Xamarin MVP, frequent speaker, author, and consultant based in Spring, Texas, USA. He has been working with the .NET framework for over 15 years, and with Xamarin since its early beta stages. He is the President of the North Houston .NET Users Group and a co-lead of the Houston Xamarin Meetup.

Jason's company, PostDotNet Consulting (postdotnet.com), is a Xamarin Consulting Partner.

Thanks to Vicki for everything, and to Jacob, Jonah, and Maggie for not bothering daddy while he was in the office.

Joe Dan Galyean is the Vice President of Application Development for Cinemark USA. He has been working in the field of software development for 14 years, primarily with .NET technologies.

Paul F. Johnson has many years of experience in cross-platform development. He started his programming career back in the 1980s on 8-bit machines. With a background in chemistry and working experience in education, he developed a large number of applications for students as well as developed code in Fortran for his Master's degree. During this time, he became interested in .NET, especially C# due to its similarities with the other C languages. By chance, a small start-up in the US called Ximian had started to develop an open source implementation of the .NET standard, so being a Linux chap was now no longer a barrier.

Over the years, this interest grew and so did Ximian. Novell bought them out, and after Novell was sold, Xamarin was formed and mobile development was the next stage.

After completing education, Paul worked on the ill-fated WowZapp messenger application followed by a number of other mobile apps on both iOS and Android. All the code for these platforms was in C#, so the idea of cross-platform mobile development was certainly going strong.

In April 2013, Paul began working on a complete rewrite of F-Track Live so that it would run on Android, iPhone, and Windows Mobile. During that time, he was contacted by Xamarin to check whether he was willing to come on board as part of the documentation team.

Paul has written *Xamarin Mobile Application Development for iOS*, Packt Publishing, and is part way through writing a new book for Packt – again using Xamarin for cross-platform development. He also has a third book in discussion with Packt Publishing for using the Xamarin Mobile platform to create an interactive adventure (including the AI aspects required for it).

I would like to dedicate my work to my ever loving wife, Becki, and to my extremely good coffee-making son, Richard, who have helped me with the phrasing of comments and supply of coffee during the technical review process. I'd also like to thank Mary and Puja at Packt Publishing for being great to work with as well as Mark (the author of this book) for showing me a trick or two and accepting criticism when given.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Xamarin and Mono – a Pathway to the Unnatural	7
Understanding Mono	7
The Xamarin product suite	8
Evaluating whether Xamarin is the right tool	9
Learning C#	11
Installing Xamarin	11
Installing Xamarin on OS X	12
Installing Xamarin on Windows	12
Development environments	12
Using the Xamarin Studio environment	12
Using Xamarin Studio to develop Android apps	13
Using Xamarin Studio to develop iOS apps	14
Using the Visual Studio environment	14
Using Visual Studio to develop Android apps	15
Using Visual Studio to develop iOS apps	15
Comparing IDEs	16
Version control	17
Summary	18
Chapter 2: Demystifying Xamarin.iOS	19
Xamarin.iOS and Ahead-of-Time compilation	19
Understanding Mono assemblies	20
Xamarin.iOS bindings	20
The design principles	20
C# types and type safety	21
Use of inheritance	21

Table of Contents

Mapping Objective-C delegates	22
Via .NET events	22
Via .NET properties	23
Via strongly typed delegates	23
Via weakly typed delegates	24
Creating binding libraries	25
Memory management	25
Disposing of objects	27
Keeping objects around	27
Limitations of using the AOT compilation	27
Runtime features disabled	28
Generating code for XIB and storyboard files	29
Generated classes	29
Designer files	29
Non-designer files	30
Outlets properties	31
Action properties	31
Xamarin.iOS Designer	31
Summary	32
Chapter 3: Demystifying Xamarin.Android	33
Mono CLR and Dalvik VM – working side by side	34
Introducing the Java Native Interface	34
Peer objects	35
Xamarin.Android application packaging	36
Understanding Mono assemblies	36
Xamarin.Android bindings	36
The design principles	37
Properties	37
Events versus listeners	38
Special help with collections	38
Interfaces	39
Mapping nested classes	40
Mapping the Runnable interface	40
Enumerations	40
Resources	41
Attributes for the ApplicationManifest.xml file	41
Editor for the ApplicationManifest.xml file	42
Garbage collection	43
JNI global and weak references	44
Mono collections	44

Table of Contents

Automatic collections	45
Helping the GC	45
Xamarin.Android Designer	46
Summary	46
Chapter 4: Developing Your First iOS App with Xamarin.iOS	47
The sample national parks app	48
Creating the sample app	48
The Project Options view	51
Running and debugging within Xamarin Studio	52
Extending the sample app	56
Storing and loading national parks	56
Adding Json.NET	57
Creating an entity class	57
Adding a JSON-formatted file	58
Loading objects from a JSON-formatted file	59
Saving objects to a JSON-formatted file	60
Running the app	60
Enhancing the UI	60
Touring the Xamarin.iOS Designer	62
Adding EditViewController and segues	65
Implementing the DoneClicked event handler	67
Implementing the DeleteClicked action	68
Passing data	69
Running the app	72
Finishing the sample app	72
Finishing DetailViewController	73
Finishing EditViewController	74
Running the app	76
MonoTouch.Dialog	76
Summary	77
Chapter 5: Developing Your First Android App with Xamarin.Android	79
The sample app	80
Creating NationalParks.Droid	80
Reviewing the app	82
Resources	82
The Resource.designer.cs file	82
The MainActivity.cs file	82
The Main.axml file	83
Project Options	84
Xamarin Studio Preferences	84

Table of Contents

Running and debugging with Xamarin Studio	84
Running apps with the Android Emulator	85
Running apps on a physical device	87
Running apps with Genymotion	87
Extending NationalParks.Droid	88
Storing and loading national parks	88
Adding Json.NET	88
Borrowing the entity class and JSON file	89
Creating the NationalParksData singleton	89
Enhancing MainActivity	91
Adding a ListView instance	91
Creating an adapter	92
Adding the New action to the ActionBar	94
Running the app	96
Creating the DetailActivity view	96
Adding ActionBar items	97
Populating DetailActivity	98
Handling the Show Photos action	98
Handling the Show Directions action	99
Adding navigation	99
Running the app	100
Creating EditActivity	100
Adding ActionBar items	101
Creating reference variables for widgets	102
Populating EditActivity	102
Handling the Save action	103
Handling the Delete action	104
Adding navigation	105
Refreshing ListView in MainActivity	106
Running the app	106
Working with Xamarin.Android projects in Visual Studio	107
Reviewing the generated elements	107
Peer objects	107
The AndroidManifest.xml file	108
The APK file	108
Summary	110
Chapter 6: The Sharing Game	111
Sharing and reuse	111
Old school source file linking	112
Creating a shared library project	112
Updating NationalParks.Droid to use shared files	115
Updating NationalParks.iOS to use shared files	116

Table of Contents

Portable Class Libraries	118
Creating NationalParks.PortableData	118
Implementing IFileHandler	119
Updating NationalParks.Droid to use PCL	120
Updating NationalParks.iOS to use PCL	121
The pros and cons of the code-sharing techniques	122
Summary	123
Chapter 7: Sharing with MvvmCross	125
Introducing MvvmCross	126
The MVVM pattern	126
Views	127
ViewModels	127
Commands	127
Data binding	128
The binding modes	129
The INotifyPropertyChanged interface	129
Binding specifications	130
Navigating between ViewModels	131
Passing parameters	131
Solution/project organization	132
The startup process	132
Creating NationalParks.MvvmCross	134
Creating the MvvmCross core project	134
Creating the MvvmCross Android app	135
Reusing NationalParks.PortableData and NationalParks.IO	137
Implementing the Android user interface	138
Implementing the master list view	139
Implementing the detail view	143
Implementing the edit view	146
Creating the MvvmCross iOS app	149
Implementing the iOS user interface	150
Implementing the master view	150
Implementing the detail view	151
Implementing the edit view	153
Considering the pros and cons	155
Summary	155
Chapter 8: Sharing with Xamarin.Forms	157
An insight into the Xamarin.Forms framework	157
Pages	158
Views	158
Layouts	159

Table of Contents

Cells	159
Navigation	160
Defining Xamarin.Forms user interfaces	160
Extensible Application Markup Language (XAML)	161
Code-behind classes	162
Data binding	164
Using Renderers	165
Native features and the DependencyService API	165
App startup	166
Shared App classes	166
iOS apps	166
Android apps	167
Project organization	167
Creating the NationalParks Xamarin.Forms app	168
Creating the solution	168
Adding NationalParks.PortableData	169
Implementing ParksListPage	170
Implementing ParkDetailPage	172
Using DependencyService to show directions and photos	174
Implementing ParkEditPage	177
Considering the pros and cons	179
Summary	180
Chapter 9: Preparing Xamarin.iOS Apps for Distribution	181
Preparing for distribution	181
Profiling Xamarin.iOS apps	182
iOS Application (Info.plist) settings	182
iOS Build settings	183
SDK Options	183
Linker Options	184
Debugging options	186
Code generation options	186
Distributing Xamarin.iOS apps	187
The Ad Hoc and enterprise distributions	187
TestFlight distribution	189
App Store submission	191
Summary	191
Chapter 10: Preparing Xamarin.Android Apps for Distribution	193
Preparing for a release APK	193
Profiling Xamarin.Android apps	194
Disabling debug	194
Changing the settings in AndroidManifest.xml	194
Changing the settings in AssemblyInfo.cs	194

Table of Contents

Android Application (AndroidManifest.xml) settings	195
Linker Options	196
Overriding the linker	197
Supported ABIs	199
Publishing a release APK	200
Keystores	200
Publishing from Xamarin.Android	200
Republishing from Xamarin.Android	203
Publishing from Visual Studio	203
App distribution options	203
Summary	203
Index	205

Preface

Mobile applications have revolutionized the way we communicate and interact with each other, and their development is now beginning to reach a certain level of maturity. To support the development of mobile apps, there are now many tools and environments available.

Xamarin is a toolset that has seen increasing success in recent years and is gaining more and more interest, particularly from development shops that have a significant investment in .NET and C# resources. Xamarin wraps each platform's native APIs with a C# wrapper, allowing the developer to interact with the environment in essentially the same way as any native developer would. As Xamarin apps are developed in C#, a whole new possibility of sharing code across platforms comes into play with all the associated benefits and challenges.

As companies look to adopt Xamarin, new Xamarin developers will be required; where do they come from? In many cases, there will be existing seasoned mobile developers who are already familiar with Android and iOS development.

That's where this book comes in; the idea being to provide a quick path for developers already familiar with Android and/or iOS development so they can get up to speed with Xamarin development. To that end, this book does not focus on the basics of developing Android and iOS apps; rather, we focus on teaching experienced Android and iOS developers how to develop apps using Mono, C#, and the Xamarin suite of tools. Specifically, we focus on the following topics:

- **Architecture:** This explains how the Xamarin products allow the use of Mono and C# to develop Android and iOS apps
- **Tools:** This describes the tools provided to support the development of applications

- Code sharing: This explains the types of code that can be shared between Android and iOS apps and the issues that might arise
- Distribution: This explains the special considerations that should be made when distributing Xamarin.Android and Xamarin.iOS apps

It should be noted that sample apps and code snippets are provided where appropriate.

When I first started using C# to develop iOS apps, it just felt a little strange. I was no fan of Objective-C, but when did C# become the cross-platform tool of choice? I always had a lot of respect for what the Mono team accomplished, but I generally had the view that Microsoft would eventually prohibit C# and .NET from being terribly successful on any platform that they did not own. Being a Star Wars fan, and somewhat of a geek, I was reminded of a conversation from Episode III. If you recall a certain scene between Anakin and Palpatine, where Anakin realizes Palpatine knew the dark side of the force; just replace the dark side of the force with Xamarin and you get Palpatine turning to you saying: "Xamarin is a pathway to many abilities some consider to be unnatural." That's pretty much the feeling I had; was I selling out to learn a cross-platform set of technologies that would eventually completely tie me to Windows?

Two years later, I feel fairly comfortable answering that question as no! Obviously, we work in a dynamic industry and things can change in an instant, but the technology world is in a different place than it was 10 years ago, and cross-platform C# and .NET seem to play in Microsoft's favor now. So, the strange feeling has been diminished with success, and seeing how the relationship between Microsoft and Xamarin has only gone from strength to strength, I am encouraged.

If you are coming from an Objective-C or Java background, you will likely have the same feelings from time to time, but if you give the tools a chance I think you will be amazed.

I hope that you find this book a valuable resource on your path to becoming a productive mobile application developer with the Xamarin suite of products.

What this book covers

Chapter 1, Xamarin and Mono – a Pathway to the Unnatural, provides an overview of the Mono project and the suite of Mono-based commercial products offered by Xamarin.

Chapter 2, Demystifying Xamarin.iOS, describes how Mono and the iOS platform coexist and allow developers to build iOS apps using C#.

Chapter 3, Demystifying Xamarin.Android, describes how Mono and the Android platform coexist and allow developers to build Android apps using C#.

Chapter 4, Developing Your First iOS App with Xamarin.iOS, walks you through the process of creating, compiling, running, and debugging a simple iOS app.

Chapter 5, Developing Your First Android App with Xamarin.Android, walks you through the process of creating, compiling, running, and debugging a simple Android app.

Chapter 6, The Sharing Game, presents various approaches of sharing code between Xamarin.iOS and Xamarin.Android apps.

Chapter 7, Sharing with MvvmCross, walks you through the use of the Xamarin.Mobile app, which provides a cross-platform API to access location services, contacts, and the device camera.

Chapter 8, Sharing with Xamarin.Forms, walks you through the basics of using the MvvmCross framework to increase code reuse between platforms.

Chapter 9, Preparing Xamarin.iOS Apps for Distribution, discusses various methods of distributing iOS apps, and walks you through the process of preparing a Xamarin.iOS app for distribution.

Chapter 10, Preparing Xamarin.Android Apps for Distribution, discusses various methods of distributing Android apps, and walks you through the process of preparing a Xamarin.Android app for distribution.

What you need for this book

This book contains both Android and iOS examples. The simplest configuration to create and run all the examples is to have an Intel-based Mac running OS X 10.8 (Mountain Lion) or a later version with Xcode, the iOS SDK 7.x, and Xamarin installed. The 30-day trial edition of Xamarin can be used as it installs both Xamarin.iOS and Xamarin.Android by default.

The following points provide detailed requirements based on specific features and configurations.

To create and execute the iOS examples in this book, you will need the following:

- An Intel-based Mac running OS X 10.8 (Mountain Lion) or a higher version
- Xcode and the iOS SDK 7 or a newer version installed

- Xamarin.iOS installed; the 30-day trial edition can be used
- An iPad or iPhone can be helpful, but is not essential

To use the Visual Studio plugin for Xamarin.iOS, you will need the following:

- A PC running Windows 7 or a higher version
- Visual Studio 2010, 2012, or 2013 installed; any non-Express edition
- Xamarin.iOS installed; the 30-day trial edition can be used
- Network connectivity to a Mac, which meets the requirements for compiling and running the apps

To create and execute the iOS examples in this book, you will need the following:

- A PC running Windows 7 or a higher version, or an Intel-based Mac running OS.X 10.8 (Mountain Lion) or a higher version
- Xamarin.Android installed; the 30-day trial edition can be used. Xamarin.Android includes the Android SDK
- An Android phone or tablet can be helpful, but is not essential

To use the Visual Studio plugin for Xamarin.Android you will need the following:

- A PC running Windows 7 or a higher version
- Visual Studio 2010, 2012, or 2013; any non-Express edition
- Xamarin.Android installed; the 30-day trial edition can be used. Xamarin.Android includes the Android SDK

Who this book is for

This book is a great resource for mobile developers who are already familiar with Android and/or iOS development and need to get up to speed with Xamarin development quickly. It is assumed that you have a background of Android, iOS and C#. The book provides an overview of the Xamarin architecture, walks you through the process of creating and running sample apps, demonstrates the use of tools provided by Xamarin, and discusses special considerations for preparing apps for distribution.

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: "Set a breakpoint on the `SetContentView()` statement."

A block of code is set as follows:

```
protected string GetFilename()
{
    return Path.Combine (
        Environment.GetFolderPath(
            Environment.SpecialFolder.MyDocuments),
        "NationalParks.json");
}
```

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: "From the **Project** menu, select **Publish Android Project**."



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 through the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

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/support>, 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 to our website, or added to any list of existing errata, under the Errata section of that title.

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

Xamarin and Mono – a Pathway to the Unnatural

This chapter provides an overview of the Mono project and the suite of Mono-based commercial products offered by Xamarin. To begin this pathway into the unknown, this chapter will cover the following topics:

- Understanding Mono
- Why you should use Xamarin
- Installation of Xamarin.Android and Xamarin.iOS
- Using Xamarin Studio and Visual Studio for development
- Options for source control

Understanding Mono

Before we jump straight into a conversation about Xamarin, let's discuss a related topic: Mono. Mono is an open source cross-platform implementation of the .NET platform. This includes a **Common Language Runtime (CLR)** that is binary compatible with Microsoft .NET, a set of compilers for languages such as C#, and an implementation of the .NET runtime libraries. The Mono CLR has been ported to many platforms, which include the Linux and BSD-based operating systems (which are not limited to just Android, iOS, and OS X), Windows-based systems, and even some game consoles such as the Wii, Xbox 360, and PS4.

The Mono project was started by a Ximian, which was purchased by Novell. Xamarin now leads Mono.

The Xamarin product suite

Xamarin is a software company that offers a suite of commercial Mono-based products that allow developers to create apps for Android, iOS, and OS X using C# and the .NET framework. Xamarin's cofounder, Miguel de Icaza, has directed the Mono project since its inception in 2001.

Xamarin's primary product offerings are:

- Xamarin.Android (formerly Mono for Android)
- Xamarin.iOS (formerly MonoTouch)
- Xamarin.Mac

Each of these products is licensed through an annual subscription with the following levels being available:

- **Starter:** This subscription is actually free, but restricts the size of apps.
- **Indie:** This subscription provides everything needed to build and deploy full-featured mobile apps, but can only be purchased by companies with five or less employees. This subscription also does not include the use of the Visual Studio add-in discussed in the *Using the Visual Studio environment* section.
- **Business:** This subscription adds the use of the Visual Studio add-in as well as e-mail support.
- **Enterprise:** This subscription adds access to a set of prime components, hotfixes, and enhanced support.



A link for the pricing information can be found at the end of this section.
For quick reference, please visit <https://store.xamarin.com>.

Xamarin also hosts a component store; a market place to buy and sell components that can be used in Xamarin apps. The components in this store can be distributed for free or sold, and Xamarin pays component vendors a percentage of the revenue collected from the sales.

Another service that Xamarin offers is the Test Cloud. As the name implies, this is a cloud-based testing service that allows teams to create automated testing capabilities for their apps that can be run against a plethora of physical devices. This is particularly important for Android developers as there are far more devices that need to be considered.

The following table provides useful links to additional information about the Xamarin suite:

Type of information	URL to access it
Product information	http://xamarin.com/tour http://xamarin.com/csharp http://xamarin.com/products http://xamarin.com/faq
Product pricing	https://store.xamarin.com
Component store	https://components.xamarin.com
Xamarin Test Cloud	http://xamarin.com/test-cloud

Evaluating whether Xamarin is the right tool

Now that you have some background on Mono and the Xamarin suite of products, you might want to ask yourself: "Is Xamarin the right tool for my project?"

The benefits of using Xamarin are as follows:

- **It builds on your existing skills of using C# and .NET:** Because of the huge number of features available to both the C# language and the .NET framework, it requires a huge investment of time and energy for developers to master them. Although you can argue that Java and Objective-C have similarities (being object-oriented languages), there is a real cost associated with transferring your proficiency in C# and .NET to make the same claim regarding Java or Objective-C. This is where Xamarin comes to your rescue; individuals and groups that have made a significant investment in C# and .NET might turn to it if they wish to develop iOS and Android apps due to the requirement of these skills.

- **Allows for reusability of code in cross-platform development:** Although the Xamarin suite prevents you to create an app that can also be deployed to Android, iOS, and WP8, it compensates for this by providing you with the capability to recycle huge portions of your code base across all of these platforms. The general process that makes this all so much easier for you is that the user interface code and the code that deals with the device capabilities tend to be written for each platform. With this, things such as client-side logic (proxies and caching), client-side validation, data caching, and client-side data storage can potentially be reused, saving you a huge amount of time and energy. I have personally seen Android and iOS apps share as much as 50 percent of the code base and some report as high as 80 percent. The more you invest in the approach to reuse, the more likely you will achieve a higher percentage.

However, there are some drawbacks when it comes to using Xamarin:

- **Costs due to licensing requirements:** The Xamarin suite or tools are all commercial tools and must be licensed, meaning there is a tangible cost of entry. You can check Xamarin's website for the current pricing.
- **Waiting for updates:** You will find that there is some lag time between a new release of a platform (Android/iOS) and the corresponding release of the Xamarin products that support it. Xamarin has done a great job of releasing Xamarin.iOS on the same day when the new versions of the OS are made available. Xamarin.Android generally lags behind because Google does not make beta/preview versions available. In some ways, this delay is not a big issue at least for phone apps; the telecoms generally take some period of time before they provide the newest Android versions to be downloaded.
- **Performance and memory management:** This is probably more of a concern for Xamarin.Android than Xamarin.iOS. As you will see in *Chapter 2, Demystifying Xamarin.iOS*, Xamarin.iOS essentially builds a binary executable much like those produced by using just Xcode and the iOS SDK. However, as we will see in *Chapter 3, Demystifying Xamarin.Android*, Xamarin.Android relies on deploying the Mono CLR and the communications between the Mono CLR and the Dalvik VM. In some cases, Xamarin.Android will allocate Java and C# objects to achieve some of the "magic" and "sorcery" behind developing in C# or .NET on an Android device. As a result of this, Xamarin.Android will affect both the memory footprint and execution performance.

- **Distribution size:** There are a number of runtime libraries that must be distributed or linked with Xamarin apps. A discussion of the actual size and strategies to minimize the distribution size is reserved for the last chapter.

While the list of drawbacks might seem extensive, in most cases, the impact of each can be minimized. I have chosen to build out a Xamarin consulting practice because I place a high value on the benefits identified and feel like many groups that have a significant investment in C# and .NET will see the same value. If you are a group or an individual that places a great value on Xamarin's benefits, then you should certainly consider using it.

Learning C#

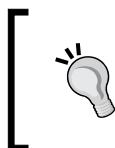
This book assumes that you have a working knowledge of C# and .NET. Since this might not be the case for some readers, we have included a few links to help you get up to speed. Xamarin provides the following link which presents C# from an Objective-C perspective: http://docs.xamarin.com/guides/ios/advanced_topics/xamarin_for_objc/primer

Microsoft provides a set of tutorials for learning C# available at: [http://msdn.microsoft.com/en-us/library/aa288436\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa288436(v=vs.71).aspx).

Installing Xamarin

Before moving on, we need to install Xamarin. This section will show you the steps to install Xamarin on both the Android and iOS platforms, notably Xamarin.Android and Xamarin.iOS, on both OS X and Windows.

Since Xamarin.iOS is dependent on the latest iOS SDK and the latest Xcode, both of these should be installed prior to starting the OS X install.



Both Xcode and the iOS SDK are free and you can download these installs from: <https://developer.apple.com/devcenter/ios/index.action#downloads>.

Also, note that you can install Xcode from the OS X App Store.

Likewise, Xamarin.Android is dependent on the latest Android SDK; however, the difference being that the Xamarin install will automatically download the Android SDK and install it as part of the overall install process. So, no separate steps need to be taken. If you already have installed the Android SDK, you have just been handed the opportunity to use it.

Installing Xamarin on OS X

To install Xamarin on OS X, go to www.xamarin.com, download the OS X installer to launch it, and follow the directions. Be sure to choose to install both Xamarin.Android and Xamarin.iOS; Xamarin.Mac is optional.

The Xamarin.iOS Visual Studio plugin uses the build server called mtbserver to compile the iOS code on your Mac. If you plan to use the Visual Studio plugin, be sure to choose to allow network connections.

Installing Xamarin on Windows

Now, we move on to the Windows installation process. If you plan on using the Visual Studio add-in, Visual Studio will need to be installed prior to installing Xamarin.

To install Xamarin on Windows, you need to visit www.xamarin.com, download the Windows installer, launch it, and then follow the directions. Be sure to install both Xamarin.Android and Xamarin.iOS.

Development environments

Developers have two options when it comes to IDEs: Xamarin Studio or Visual Studio. This section will show you how to develop apps for both Android and iOS through both of these studios.

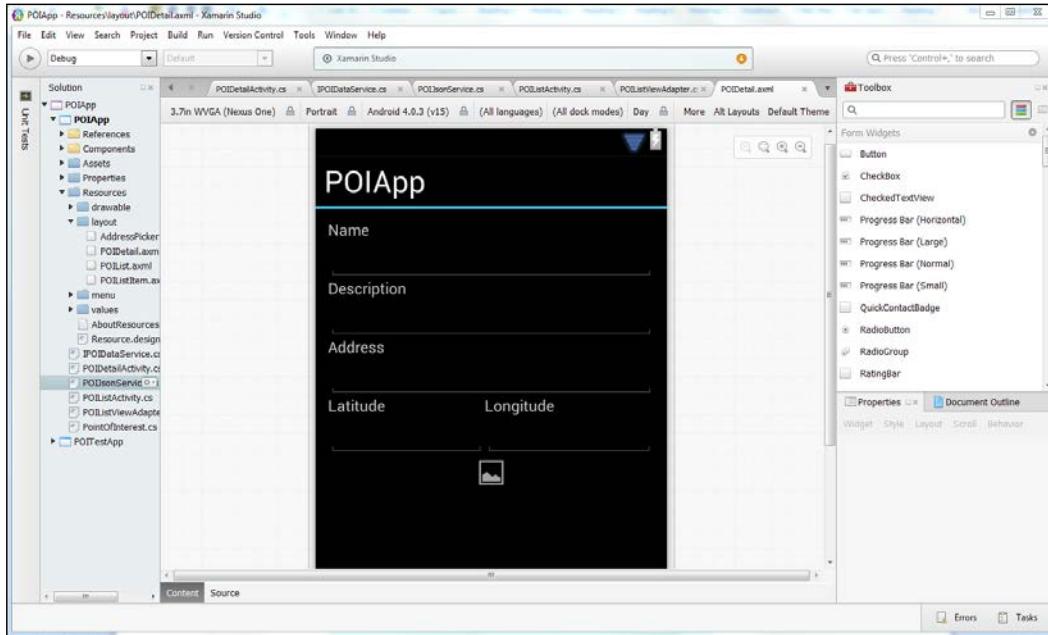
Using the Xamarin Studio environment

Xamarin Studio is a customized version of the MonoDevelop IDE and this can be used to develop applications for Android, iOS, and OS X. Xamarin Studio is available on both OS X and Windows with highly advanced and useful features such as:

- Code completion
- Smart syntax highlighting
- Code navigation

- Code tooltips
- Integrated debugging for mobile apps running in emulators or on devices
- Source control integration with Git and Subversion built-in

If you look at the following screenshot, you will see how Xamarin Studio is shown with the Android user interface designer opened:



Using Xamarin Studio to develop Android apps

Xamarin Studio and the Xamarin.Android add-in allow the complete development and debugging of Android apps without use of any other IDEs. The Android UI designer can also be used from within Xamarin Studio.

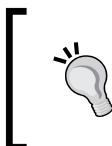
Using Xamarin Studio to develop iOS apps

Xamarin Studio and the Xamarin.iOS add-in allow the development and testing of iOS apps when installed on a Mac with Xcode and the iOS SDK. All code can be written, compiled, and debugged from within Xamarin Studio. In general, the user interface XIB and/or storyboard files must be built within Xcode; Xamarin Studio provides a link to Xcode such that when a xib or storyboard file is double-clicked on, Xcode will be launched.

There is a caveat to this; Xamarin has an iOS UI designer built for Xamarin Studio, yet it has remained in an alpha status for almost a year. I have seen a number of posts on various forums indicating it is stable and safe to use, but Xamarin has been slow to clarify why it is still in alpha status and when it will move to a stable status. We will discuss the use of the iOS UI designer in more detail in *Chapter 4, Developing Your First iOS App with Xamarin.iOS*.

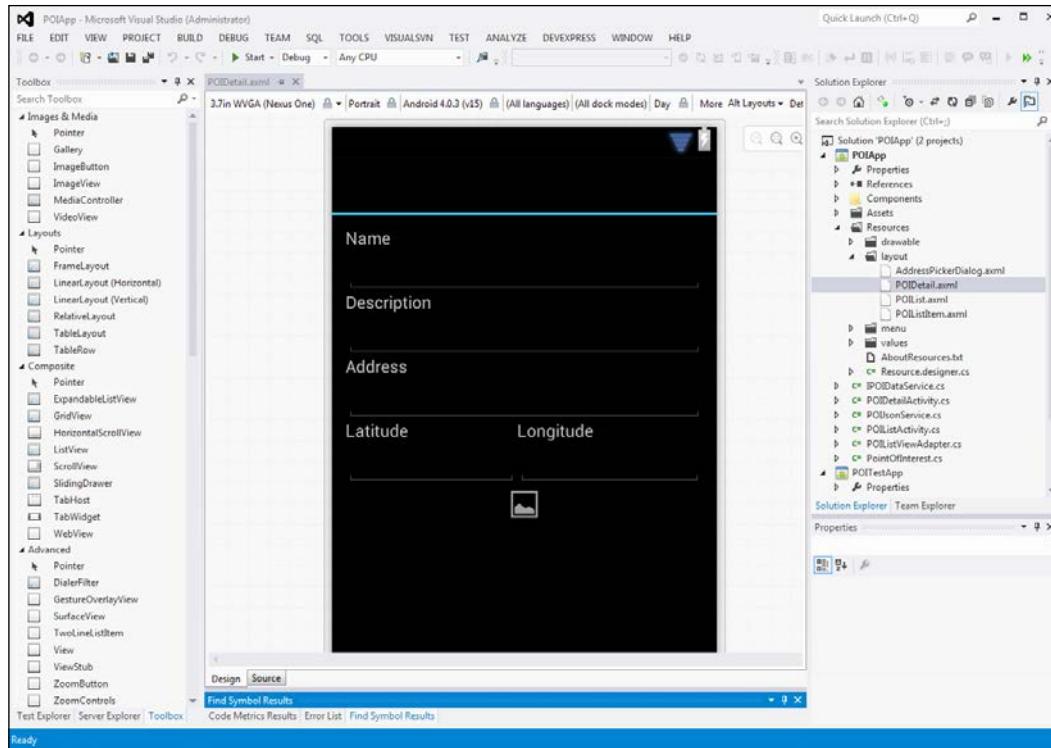
Using the Visual Studio environment

Xamarin for Visual Studio is an add-in that supports the development of the Xamarin.Android and Xamarin.iOS apps and is available to business and enterprise subscribers. This add-in can be used with any non-Express edition of Visual Studio 2010 through to Version 2013. Android apps can be completely developed using Visual Studio. In order to develop iOS apps, you will still need a Mac with the iOS SDK and Xcode to compile and create the user interface xib and/or storyboard files.



If you already have a license for Visual Studio and are comfortable with the environment, this add-in will be better suited to you than Xamarin Studio due to it being simple to use.

The following screenshot shows Visual Studio 2012 with the Android user interface designer opened:



The Android user interface designer

Using Visual Studio to develop Android apps

The Visual Studio add-in for Xamarin.Android allows the full development and debugging of Android apps without the use of any other IDE. This add-in provides the usage of the Android UI designer from within Visual Studio. For those that have the appropriate licenses and are comfortable with Visual Studio, this might be the best option for Android development.

Using Visual Studio to develop iOS apps

The Visual Studio add-in for Xamarin.iOS allows you to develop and test iOS apps, but only in conjunction with the use of a Mac with both Xcode and the iOS SDK installed. The iOS code must be compiled and executed on a Mac using the Xamarin's mtbserver. Xcode on a Mac must also be used to develop the user interface xib and/or storyboard files for an iOS app. We will discuss this configuration in more detail in *Chapter 4, Developing Your First iOS App with Xamarin.iOS*.



Solution and project files created and used by Xamarin Studio are completely compatible with Visual Studio. This gives teams the flexibility to choose which IDE to use and they can easily change throughout the duration of a project.



Comparing IDEs

The advantages and disadvantages of adopting each IDE are shown in the following table:

IDE	Pros	Cons
Xamarin Studio	Available for all Xamarin subscription levels Runs on Windows and OS X	Limited number of productivity add-ins available Does not offer support for the use of TFS for source control
Visual Studio	Most C# developers are already familiar and comfortable with Visual Studio Allows the use of popular productivity add-ins such as ReShaper and CodeRush Allows the use of TFS for source control and issue tracking	Requires a business or enterprise subscription of Xamarin Requires a license of VS Runs on Windows only For iOS development, requires a more complex configuration in which VS must communicate with a Mac running Xcode to perform builds and UI development must be done with Xcode

Version control

Version control can be a challenge anytime you have a diverse set of development tools, and Xamarin certainly adds diversity to most shops. The challenge is making it easy to share and manage code from all of the different IDEs and client apps that folks will be working with; many times they do not have access to the same repositories. Since the benefits of using Xamarin are very attractive to existing .NET shops, many Xamarin developers will find themselves working in environments already committed to using Microsoft **Team Foundation Server (TFS)**. Unfortunately, it's not always easy to connect to TFS from non-Microsoft tools. In the case of Xamarin Studio, there is an open source add-in that cannot be directly supported by Xamarin and can be challenging to configure.

Other version control systems to consider include Git and Subversion. Xamarin Studio contains built-in support for both Git and Subversion, and add-ins for both of these tools exist for Visual Studio. The following table contains useful URLs to download and read about the various add-ins:

Add-in	URL to access it
TFS add-in for Xamarin Studio	https://github.com/Indomitable/monodevelop-tfs-addin http://www.teamaddins.com/blog
Git for Visual Studio	(VS2013 has built-in support) http://msdn.microsoft.com/en-us/library/hh850437.aspx (VS2012 requires a free plugin) http://visualstudiogallery.msdn.microsoft.com/abafc7d6-dcaa-40f4-8a5e-d6724bdb980c
Subversion add-in for Visual Studio (by VisualSVN)	http://www.visualsvn.com/visualsvn/?gclid=CMmSnY-opL0CFa07MgodDksA5g

Like many aspects of software development, there is not a "one size fits all". The following table outlines some of the pros and cons to consider when deciding on a source control solution for Xamarin projects:

VCS Tool	Pros	Cons
TFS	Already in use by many shops that will consider Xamarin. Free add-in for Xamarin Studio.	Xamarin Studio add-in has been known to be problematic to use in the past.
Git	Built-in support in Xamarin. Free add-in available for Visual Studio 2012 and 2013.	Difficult to share and synchronize code with other teams in a large organization that might be using TFS for their C# code.
Subversion	Built-in support in Xamarin Studio. Commercial add-in for Visual Studio.	Difficult to share and synchronize code with other teams in a large organization that might be using TFS for their C# code.

If you already have a significant investment in using TFS, try to make that work for your Xamarin development as well. This can be done by either having developers use Visual Studio or trying your luck with the TFS add-in for Xamarin Studio.

Summary

In this chapter, we provided an introduction to Mono and the suite of commercial products offered by Xamarin and considered the pros and cons of using Xamarin. We also went through the installation process and took a first look at the IDE options available to developers.

In the next chapter, we will take a look at the architecture of the Xamarin.iOS product.

2

Demystifying Xamarin.iOS

Now that we have a little background on Mono and Xamarin, let's dive in and see how Xamarin.iOS works. This chapter covers the following topics:

- Xamarin.iOS and AOT compilation
- Mono assemblies
- Xamarin.iOS bindings
- Memory management for Xamarin.iOS apps
- XIB and storyboard code generation
- Xamarin.iOS Designer

Xamarin.iOS and Ahead-of-Time compilation

Unlike most Mono or .NET apps, Xamarin.iOS apps are statically compiled, where compilation is accomplished through the Mono **Ahead-of-Time (AOT)** compilation facilities. AOT is used to comply with Apple's requirements, for example, the use of iOS apps to compile, refraining from Just-in-Time compilation facilities, or running on virtual machines.

Use of AOT compilation comes with some limitations regarding the C# language. These limitations will be easier to discuss after discussing the approach to binding iOS to C# and .NET. This is why we have pushed this topic to the *Limitations of using the AOT compilation* section in the later part of this chapter.



Additional information about Mono AOT compilation can be found at the following link:

http://www.mono-project.com/AOT#Full_AOT



Understanding Mono assemblies

Xamarin.iOS ships with an extended subset of Silverlight and desktop .NET assemblies. These libraries provide the .NET runtime library support for developers, including namespaces such as `System.IO` and `System.Threading`.

Xamarin.iOS is not binary compatible with assemblies compiled for a different profile, meaning your code must be recompiled to generate assemblies that specifically target the Xamarin.iOS profile. This is essentially the same thing you have to do if you are targeting other profiles such as Silverlight or .NET 4.5.



For a complete list of assemblies that ship with Xamarin.iOS, please refer to the following link:

http://docs.xamarin.com/guides/ios/under_the_hood/assemblies



Xamarin.iOS bindings

In this section, you will discover one of the main sources of power behind Xamarin.iOS. This ships with a set of binding libraries that provides support for iOS development. What will follow are some details into each of these bindings.

The design principles

A number of goals or design principles guided the development of the binding libraries. These principles are critical to make C# developers productive in an iOS development. The following represents a summary of the design principles:

- Allow developers to subclass Objective-C classes in the same way as they subclass other .NET classes
- Provide a way to call arbitrary Objective-C libraries
- Transform the common Objective-C tasks into something much easier while making the difficult Objective-C tasks possible to complete

- Expose Objective-C properties as C# properties as well as expose a strongly typed API
- Use Native C# types in lieu of Objective-C types when possible
- Support both C# events and Objective-C Delegation as well as expose C# delegates to Object-C APIs



This section has provided you with a general idea of the principles to bear in mind. If you are curious to find a complete discussion, you can refer to the official documentation available at the following link:
http://docs.xamarin.com/guides/ios/under_the_hood/api_design/

C# types and type safety

The Xamarin.iOS bindings are designed to use types familiar to C# developers and to increase type safety when possible.

For example, the API uses C# string instead of `NSString`, meaning the `text` property in `UILabel` is defined in the iOS SDK in the following manner:

```
@property(nonatomic, copy) NSString *text
```

Also, this is exposed in Xamarin.iOS as follows:

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

Behind the scenes, the framework takes care of marshaling C# types to the appropriate type expected by the iOS SDK.

Another example is the treatment of `NSArray`. Rather than exposing weakly typed arrays, Xamarin.iOS exposes strongly typed arrays to the following Object-C property on `UIView`:

```
@property(nonatomic, readonly, copy) NSArray *subviews
```

This is exposed as a C# property in the following manner:

```
UIView[] Subviews { get; }
```

Use of inheritance

Xamarin.iOS allows you to extend any Objective-C type in the same manner you will extend any C# type and features like calling "base" from overridden methods work as predicted.

Mapping Objective-C delegates

In Objective-C, the delegation design pattern is used extensively to allocate responsibility to various objects. Xamarin faced a few inherent challenges in mapping iOS delegates to C# and .NET.

In Objective-C, delegates in Objective-C are implemented as objects that respond to a set of methods. This set of methods is generally defined as a protocol, and although it resembles a C# interface, there is in fact a significant difference between a C# interface and an Objective-C protocol:

- In C#, an object that *implements* an interface is required to implement all the methods defined on the interface
- On the other hand, objects in Objective-C that *adopt* a protocol are not required to implement the methods of the protocol for the given circumstance

Another challenge is that, in many ways, traditional .NET frameworks have relied more heavily on events to accomplish similar capabilities, and the event model is much more familiar to .NET developers.

With these differences in mind and hoping to make Xamarin.iOS as intuitive to C# developers as possible without compromising the iOS architecture, Xamarin.iOS provides four different ways to implement delegate functionality:

- Via .NET events
- Via .NET properties
- Via strongly typed delegates
- Via weakly typed delegates

Via .NET events

For many types, Xamarin.iOS automatically creates an appropriate delegate and forwards delegate calls to corresponding .NET events. This makes the development experience very natural to C# and .NET developers.

`UIWebView` is a good example of this. iOS defines `UIWebViewDelegate`, which contains a number of methods that a `UIWebView` will forward if a delegate is assigned that includes the following:

- `webViewDidStartLoad`
- `webViewDidFinishLoad`
- `webView:didFailLoadWithError`

What we find in the Xamarin.iOS class `MonoTouch.UIKit.UIWebView` are three events that correspond to the following methods:

- `LoadStarted`
- `LoadFinished`
- `LoadError`

Via .NET properties

Although events have the advantage of having multiple subscribers, they come with their own limitations. Specifically, this could be where events cannot have a return type. In situations where a delegate method must return a value, delegate properties are used. The following example shows you how to use a delegate property for `UITextField`. In this example, an anonymous method is assigned to the delegate property `ShouldReturn` on `UITextField`:

```
firstNameTextField.ShouldReturn = delegate (textfield)
{
    textfield.ResignFirstResponder ();
    return true;
}
```

Via strongly typed delegates

If events or delegate properties have not been provided or if you would just rather work with a delegate, you will be pleased to hear that Xamarin.iOS provides a set of .NET classes that correspond to each iOS delegate. These classes contain a definition for each method defined on the corresponding protocol. Methods that require implementations are defined as abstract and methods that are optional are defined as virtual. To use one of these delegates, a developer simply creates a new class that inherits from the desired delegate and overrides the methods that need to be implemented.

For an example of using strongly typed delegate, we will turn to `UITableViewDataSource`. This is the protocol iOS defines to populate a `UITableView` instance. The following example demonstrates a data source that can be used to populate `UITableView` with phone numbers:

```
public class PhoneDataSource : UITableViewDataSource
{
    List<string> _phoneList;
    public PhoneDataSource (List<string> phoneList) {
        _phoneList = phoneList;
    }
}
```

```
public override int RowsInSection(UITableView
    tableView, int section)
{
    return _phoneList.Count;
}
public override UITableViewCell GetCell(UITableView
    tableView, NSIndexPath indexPath) {
    ... // create and populate UITableViewCell
    return cell;
}
}
```

Now that we have created the delegate, we need to assign it to a UITableView instance. The property for the UITableViewDataSource delegate is named `Source` with the following code that shows you how to make the assignment:

```
phoneTableView.Source = new PhoneDataSource(phoneList);
```

Via weakly typed delegates

Lastly, Xamarin.iOS provides you with a way to use weakly typed delegates. Unfortunately, this method requires a bit more work for the developer.

In Xamarin.iOS, weak delegates can be created using any class that inherits from `NSObject`. When creating a weak delegate, you are being handed the responsibility to properly decorate your class using the `Export` attribute, which effectively teaches iOS how the methods are mapped. The following example shows a weak delegate with the appropriate attribute specifications:

```
public class WeakPhoneDataSource : NSObject
{
    ...
    [Export ("tableView:numberOfRowsInSection:")]
    public override int RowsInSection(UITableView
        tableView, int section)
    {
        ...
    }
    [Export ("tableView:cellForRowAtIndexPath:")]
    public override UITableViewCell GetCell(UITableView
        tableView, NSIndexPath indexPath) {
        ...
    }
}
```

The last few steps assign the weak delegate to a `UITableView` instance. By Xamarin.iOS convention, weak delegate property names always begin with `Weak`. The following example shows you how to assign the weak data source delegate:

```
phoneTableView.WeakSource =
    new WeakPhoneDataSource(...);
```

Once a weak delegate has been assigned, any assigned strong delegates will cease to receive calls.

Creating binding libraries

There might be times when you are required to create your own binding library for an Objective-C library that is not delivered as part of Xamarin.iOS and can't be found in the Xamarin component store. Xamarin provides a great deal of guidance to create bindings as well as an automated tool to help with some of the drudgery work. The following links provide guidance to create custom bindings for Objective-C libraries:

Type of information	URL to access it
General binding information	http://docs.xamarin.com/guides/ios/advanced_topics/binding_objective-c/
Use of the Objective Sharpie automation tool	http://docs.xamarin.com/guides/ios/advanced_topics/binding_objective-c/objective_sharpie/
Binding types reference	http://docs.xamarin.com/guides/ios/advanced_topics/binding_objective-c/binding_types_reference_guide/

Memory management

When it comes to releasing resources, Xamarin.iOS has this covered for you through **garbage collector** (GC), which does this on your behalf. On top of this, all objects that are derived from `NSObject` utilize the `System.IDisposable` interface so that developers have some control over it when the memory is released.

`NSObject` not only implements the `IDisposable` interface, but also follows the .NET dispose pattern. The `IDisposable` interface only requires a single method to be implemented, `Dispose()`. The dispose pattern requires an additional method to be implemented, `Dispose(bool disposing)`. The `disposing` parameter indicates whether the method is being called from the `Dispose()` method, in which case the value is `true`, or from the `Finalize` method, in which case the value is `false`.

The disposing parameter is intended to be used to determine if managed objects should be freed. If the value is `true`, the managed objects should be released. Unmanaged objects should be released regardless of the value. The following code demonstrates what should be present in the `Dispose()` method:

```
public void Dispose ()
{
    this.Dispose (true);
    GC.SuppressFinalize (this);
}
```

Take note of the call to `Dispose(bool disposing)` with a value of `true`. Conveniently, the `Dispose()` method is implemented for you by the framework as a virtual method on `NSObject`. The following code demonstrates an implementation of the `Dispose(bool disposing)` method:

```
class MyViewController : UIViewController {

    UIImage myImage;

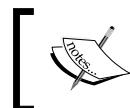
    . . .

    public override void Dispose (bool disposing)
    {
        if (disposing) {
            if (myImage != null) {
                myImage.Dispose ();
                myImage = null;
            }
        }
    }

    // Free unmanaged objects regardless of value.

    base.Dispose (disposing)

}
}
```



Again, notice the call to `base.Dispose(disposing)`. This call is very important as it deals with resources managed within the framework itself.

Why the fuss? Why not clean up everything in `Dispose()`? The answer lies in the garbage collector. The order in which the garbage collector destroys objects is not defined and thus is unpredictable and subject to change. The .NET dispose pattern helps prevent the finalizer from calling `Dispose()` on objects that have already been disposed of.



You can read more about the .NET dispose pattern at the following link:
<http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx>

Disposing of a managed object renders it useless. Even though references to the object might still exist, you need to structure your software with the assumption that an object that has been disposed of is no longer valid. In some cases, an `ObjectDisposedException` will be thrown when accessing methods of a disposed object.

Disposing of objects

Anytime you have an object that is holding substantial resources and is no longer required, call the `Dispose()` method. The GC is convenient and fairly sophisticated but might not have a complete picture as to the amount of resources a specific object has allocated, particularly if those resources are associated with unmanaged objects.

Keeping objects around

To prevent an object from being destroyed, you simply need to be sure there is at least one reference to the object maintained. Once an object's reference count reaches 0, the GC is all happy to call the `Dispose()` method on it and the object is no longer usable.

Limitations of using the AOT compilation

As we mentioned earlier in this chapter, some limitations come with the use of the AOT compilation. The following sections outline the limitations imposed by Xamarin.iOS due to the use of the AOT compilation:

- No generic subclasses of `NSObject` are allowed. The following will not be allowed since `UIViewController` is a subclass of `NSObject`:

```
class MainViewController<T> : UIViewController {
    ...
}
```

- P/Invoke is not supported in generic classes, so the following is not supported in Xamarin.iOS:

```
class MyGenericType<T> {
    [DllImport ("System")]
    public static extern int getpid ();
}
```

- `Property.SetInfo()` on a `Nullable<T>` Type is not supported. Using Reflection's `Property.SetInfo()` to set the value on a `Nullable<T>` is not currently supported.
- No dynamic code generation. The iOS kernel prevents an app from generating code dynamically and thus Xamarin.iOS imposes the following limitations:
 - Neither `System.Reflection.Emit` nor `System.Runtime.Remoting` is available
 - No support to create types dynamically
 - Reverse callbacks must be registered with the runtime at compile time
- There is a further limitation for reverse callbacks. In Mono, you can pass C# delegates to unmanaged code rather than passing a function pointer. Use of AOT imposes some limitations on this:
 - Callback methods must be flagged with the Mono attribute `MonoPInvokeCallbackAttribute`
 - Callback methods must be static; there is no support for instance methods

Runtime features disabled

The following features are disabled in Xamarin.iOS:

- Profiler
- The `Reflection.Emit` functionality
- The `Reflection.Emit.Save` functionality
- COM bindings
- The JIT engine
- The metadata verifier (since there is no JIT)

Generating code for XIB and storyboard files

The Apple Interface Builder is a designer built into Xcode that allows for visual design of a user interface. The use of the Interface Builder is optional; user interfaces can be completely built using iOS APIs. The definitions created by the Interface Builder are saved in either XIB or storyboard files with the difference being that the XIB files tend to contain a single view. Storyboards, on the other hand, contain a set of views along with the transitions or segues between the views.

Xamarin Studio works in conjunction with Xcode to support the UI design. When a storyboard or XIB file is double-clicked on within Xamarin Studio, Xcode is launched to facilitate the design of the UI. Once the changes are saved in Xcode and you switched back to Xamarin Studio, C# code is generated to support the UI design captured in Xcode. The following sections describe this process in more detail.

Generated classes

Xamarin Studio generates two files for each custom class found in an XIB file or a storyboard file, a designer file, and a non-designer file. For instance, a view controller named `LoginViewController` will cause the following files to be generated:

- `LoginViewController.cs`
- `LoginViewController.designer.cs`

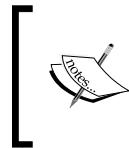
These files are generated after the changes have been saved in Xcode, and Xamarin Studio gains focus.

Designer files

Designer files contain a partial class definition for custom classes found in the XIB or storyboard file with properties being created for outlets and partial methods for the actions that are found. The following example is for a view controller with two `UITextField` controls and a single `UIButton` control:

```
[Register ("LoginViewController")]
partial class LoginViewController
{
    [Outlet]
    MonoTouch.UIKit.UITextField textPassword { get; set; }
    [Outlet]
    MonoTouch.UIKit.UITextField textUserId { get; set; }
```

```
[Action ("btnLoginClicked")]
partial void btnLoginClicked
    (MonoTouch.Foundation.NSObject sender);
void ReleaseDesignerOutlets ()
{
if (textUserId != null) {
    textUserId.Dispose ();
textUserId = null;
}
if (textPassword != null) {
    textPassword.Dispose ();
textPassword = null;
}
}
```



Designer files are automatically updated once an XIB or storyboard file has been altered. As a result, they should not be modified manually because any changes will be lost once Xamarin Studio updates them.



Non-designer files

Designer files are used alone but in conjunction with a non-designer file. The non-designer file contains a partial class specification, which completes the class defined in its corresponding designer file. The non-designer files identify the base class, defines constructors that are required to instantiate the class, and provides a place to implement functionality either by providing implementations for partial methods or by overriding virtual methods on the base class. The following example shows a non-designer file with an override and partial method implementation:

```
public partial class LoginViewController : UIViewController
{
    public LoginViewController (IntPtr handle) :
        base (handle)
}
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    // logic to perform when view has loaded...
}
partial void btnLoginClicked (NSObject sender)
{
// logic for login goes here...
}
```



Note that the partial method implementation in this file is for a method generated in the designer file in response to find an action defined in the corresponding XIB or storyboard files.

Changes made to the non-designer file will not be lost as these files are only created the first time Xamarin Studio encounters the new custom class and are not subsequently updated.

Outlets properties

Designer classes contain private properties, which correspond to outlets defined on the custom class that can then be used from the `CodeBehind` class found in the non-designer file. If you need to make these properties public, all you need to do is add the accessor properties to the non-designer file similar to how you will for any given private field.

Action properties

Designer files have the property of containing partial methods that are associated to all of the actions defined on the custom class. You should note that these methods do not contain an implementation and they serve a dual purpose:

- Their first purpose is that when you insert partial into the class body of the non-designer file, Xamarin Studio will offer to autocomplete the signatures of all non-implemented partial methods, which allows developers to implement logic for actions.
- Their other purpose is that their signatures have an attribute applied to them, exposing them to the Objective-C world. Consequently, they can be invoked once the corresponding action is triggered in iOS.

Xamarin.iOS Designer

Xamarin provides an alternative to Apple's Interface Builder. Xamarin.iOS Designer is an add-in for the Xamarin Studio environment that adds full drag-and-drop user interface design for iOS storyboards all from within Xamarin Studio. Xamarin.iOS Designer provides the following key features:

- **Compatible storyboard format:** As you will expect, Xamarin.iOS Designer creates storyboards in the same format used by Xcode and the iOS SDK, so switching back to Xcode at some point is allowed

- **Eliminates syncing with Xcode:** Using Xamarin.iOS Designer eliminates the need to use Xcode in the development process along with the synchronization problems that can occur between Xamarin Studio and Xcode
- **Easy properties:** Xamarin.iOS Designer automatically creates properties that reference controls as they are dropped on a view
- **Easy event handlers:** Xamarin.iOS Designer provides a more intuitive means to create event handlers, which work in a very similar way as Visual Studio works on other UI frameworks such as Silverlight and WPF
- **Custom controls:** User can create their own custom UI controls that are accessible from within the toolbox panel

Xamarin.iOS Designer can only be used to create storyboards. If you prefer or need to work with XIB files, you will need to continue to work with Xcode.

Summary

In this chapter, we presented the essentials of Xamarin.iOS architecture and tried to demystify the way Xamarin.iOS allows developers to create great native apps for iOS using C# and Mono. While we have obviously not covered the entire iOS SDK, we have described the approach and principles used to build Xamarin.iOS. With this knowledge in place, you should be in a good position to move forward with Xamarin.iOS development and resolve issues as they arise.

In the next chapter, we will try to accomplish the same goals for Xamarin.Android.

3

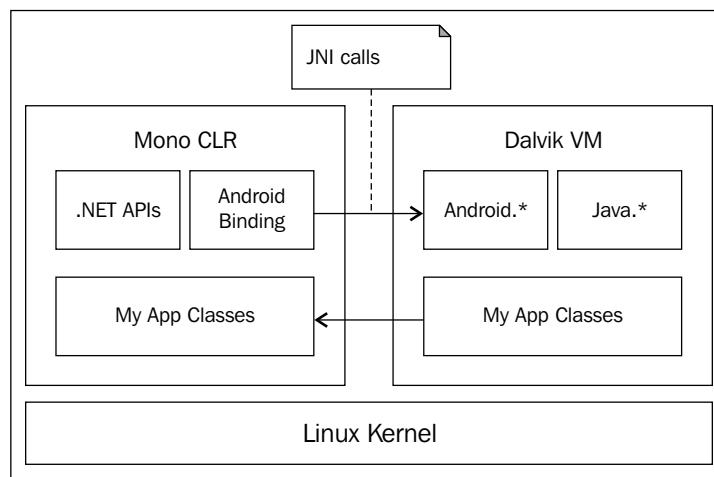
Demystifying Xamarin.Android

It's now time to take a deeper dive into Xamarin.Android to see how it pulls off the same magic as Xamarin.iOS. In this chapter, we will see that Xamarin.iOS and Xamarin.Android share many of the same design goals. However, Xamarin.Android does not rely on static compilation. Many of the goals are achieved through completely different methods. This chapter covers the following topics:

- Mono CLR and Dalvik VM – working side by side
- Application packaging
- Mono assemblies
- Xamarin.Android bindings
- Attributes for the `ApplicationManifest.xml` file
- Garbage collection

Mono CLR and Dalvik VM – working side by side

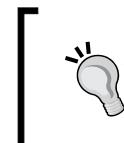
Android apps run within the **Dalvik Virtual Machine (Dalvik VM)**, which is somewhat similar to a Java VM, but optimized for devices with limited resources. As we discussed in *Chapter 1, Xamarin and Mono – a Pathway to the Unnatural*, Xamarin products are based on the Mono platform that has its own VM called the **Common Language Runtime (CLR)**. The key question to ask here is, "In which environment does a *Xamarin.Android* app run?" The answer is both. If you take a look at the next diagram, you will see for yourself how these two runtimes coexist:



Both environments seem quite different from each other, so how does an app run in both? *Xamarin.Android*'s power is achieved through a concept called peer objects and a Java framework called **Java Native Interface (JNI)**.

Introducing the Java Native Interface

Let's start with JNI. This is a framework that allows non-Java code with languages such as C++ or C#, as an example, to call or be called by Java code running inside a JVM. As you can see from the previous diagram, JNI is a critical component in the overall *Xamarin.Android* architecture.



You can find some supporting information on JNI, particularly on peer objects, in *Chapter 2, Xamarin.Android Architecture* in Packt Publishing's *Xamarin Mobile Application Development for Android*, *Mark Reynolds*.

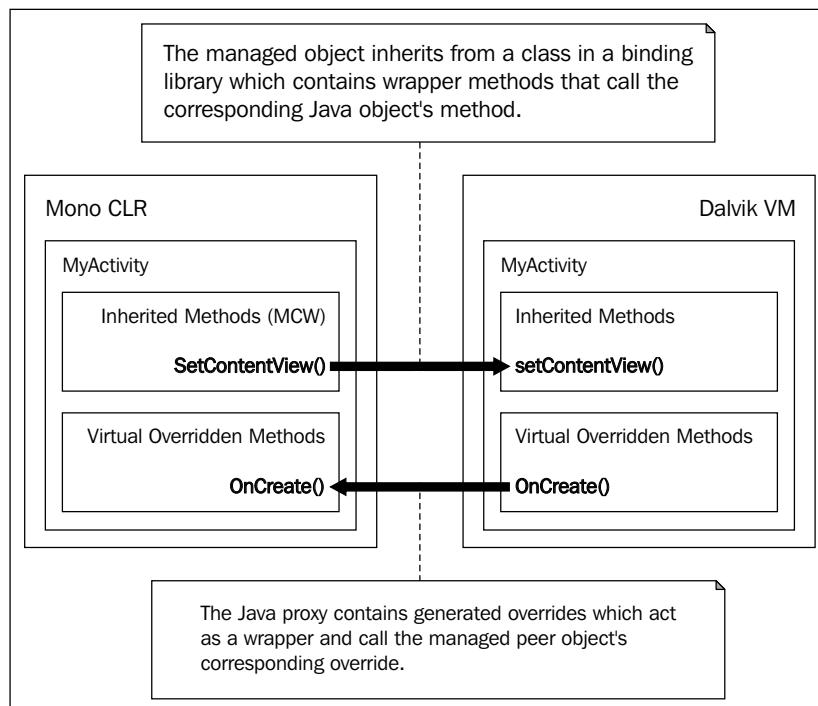
Peer objects

Peer objects are a pair of objects that work together to carry out the functionality of an Android app. One of these is a managed object residing in the Mono CLR, while the other is a Java object residing in the Dalvik VM.

Xamarin.Android is delivered with a set of assemblies called the Android binding libraries. Classes in the Android binding libraries correspond to the Java classes in the Android application framework, and the methods on the binding classes act as wrappers, to call corresponding methods on Java classes. These binding classes are commonly known as **Managed Callable Wrappers (MCW)**. Because whenever you create a C# class that inherits from one of these binding classes, a corresponding Java proxy class is generated at build time. The Java proxy contains a generated override for each overridden method in your C# class and acts as a wrapper to call the corresponding method on the C# class.

Peer objects can be created from within the Dalvik VM by the Android application framework or from within the Mono CLR by the code you write in the overridden methods. A reference between the two peer objects is kept by each instance of a MCW and can be accessed through the `Android.Runtime.IJavaObject.Handle` property.

You can see for yourself how peer objects collaborate together here:



Xamarin.Android application packaging

Android applications are delivered for installation in an Android package format, which is an archive file with a .apk extension. An Android package contains the app's code and all of the supporting files required to run the app that includes the following:

- Dalvik executables (*.dex files)
- Resources
- Native libraries
- The application manifest

Xamarin.Android apps follow the same standard with the following additions:

- C# code is compiled into assemblies and stored in a top-level folder named `assemblies`
- Mono runtime libraries are stored along with other native libraries in the `lib` folder

Understanding Mono assemblies

Like Xamarin.iOS, Xamarin.Android ships with an extended subset of Silverlight and desktop .NET assemblies. Together, these libraries provide the .NET runtime library support for developers, including namespaces such as `System.IO` and `System.Threading`.

Xamarin.Android is not binary compatible with assemblies compiled for a different profile, meaning your code must be recompiled to generate assemblies, specifically targeting the Xamarin.Android profile. This is essentially the same thing you have to do if you're targeting other profiles such as Silverlight or .NET 4.5.

[ For a complete list of assemblies that ship with Xamarin.Android, you can refer to http://docs.xamarin.com/guides/android/under_the_hood/assemblies.]

Xamarin.Android bindings

Xamarin.Android also ships with a set of binding libraries that provide the support for Android development. The binding libraries form the second big part of the magic behind Xamarin.Android similar to the way in which the Mono CLR and Dalvik VM function. The following sections delve into the details of these bindings.

The design principles

A number of goals or design principles guided the development of the binding libraries. These principles are critical to make C# developers productive in an Android development. The following represents a summary of the design principles, where you will notice some similarities with the Xamarin.iOS bindings:

- Allow developers to subclass Java classes in the same way they subclass other .NET classes
- Make common Java tasks easy, and tough Java tasks possible
- Expose JavaBean properties as C# properties
- Expose a strongly typed API
- Expose C# delegates (lambdas, anonymous methods, and `System.Delegate`) instead of single-method interfaces when appropriate and applicable
- Provide a mechanism to call arbitrary Java libraries (`Android.Runtime.JNIEnv`)

 A complete discussion around these principles can be found at http://docs.xamarin.com/guides/android/advanced_topics/api_design.

Properties

To the greatest extent possible, JavaBean properties in the Android framework classes are transformed into C# properties. The following rules are always followed whenever this takes place:

- Firstly, read/write properties are created for both the getter and setter method pairs
- Read-only properties are created for getters without the corresponding setter methods
- In the rare case that only a setter exists, no write-only properties are created
- Finally, no properties are created when the type will be an array

Events versus listeners

Android APIs follow the Java pattern in order to define and hook up event listeners. C# developers should be more familiar with the similar concepts of delegates and events.

The following is an example of Java event listeners:

```
addTicketButton.setOnClickListener (   
    new View.OnClickListener() {  
        public void onClick (View v) {  
            _ticketCount++;  
            updateLineItemCost ();  
        }  
    } );
```

This following is the equivalent code that uses C# events:

```
addTicketButton.Click += (sender, e) => {  
    _ticketCount++;  
    UpdateLineItemCost ();  
};
```

The Android bindings provide events when possible. The following rules are followed:

- When the listener has a set prefix such as `setOnClickListener`
- When the listener callback has a void return
- When the listener accepts only a single parameter, the interface has only a single method, and the interface name ends with `Listener`

When an event is not created due to one of the rules enlisted here, a specific delegate is generated that supports the appropriate signature.

Special help with collections

The native Android APIs uses list, set, and map collections extensively from `java.util`. The Android bindings expose these collections using interfaces from `System.Collections.Generic`. In addition, `Xamarin.Android` provides a set of helper classes that implement each corresponding .NET collection and provides faster marshaling because they do not actually perform a copy. The following table shows how these classes map:

Java type	.NET interface	Helper class
java.util.Set<E>	ICollection<T>	Android.Runtime.JavaSet<T>
java.util.List<E>	IList<T>	Android.Runtime.JavaList<T>
java.util.Map<K, V>	IDictionary< TKey , TValue >	Android.Runtime.JavaDictionary<K, V>
java.util.Collection<E>	ICollection<T>	Android.Runtime.JavaCollection<T>

Xamarin.Android allows you to pass any collection (which implements the correct interface) into the Android API methods. For example, `List` implements `IList` and can be used when an `IList` entity is required. However, for performance reasons, it is recommended that you use the helper classes anytime you need to pass any of these collection types into an Android API method.

Interfaces

Java and C# both support interfaces; however, Java supports additional capabilities. Also, both support the ability to define a set of method names and signatures. In addition, Java supports the following:

- Nested interface definitions
- Fields (`public final static` only)

In general, the following items describe how the Android bindings provide the following Java interfaces:

- A C# interface with the same name but prefixed by `I` and containing method declarations is created. For example, `android.view.Menu` is created as `Android.Views.IMenu`.
- An abstract class with the same name as the Java interface is generated, which contains definitions for the constants from the Android interface. For example, the constants from `android.view.Menu` are placed in the generated abstract class `Android.Views.Menu`.
- A C# interface is generated for each nested interface and is given a name prefixed by `I`, the name of the parent Java interface, followed by the name of the nested Java interface.
- Classes in the Android bindings that implement an Android interface containing constants get a nested `InterfaceConsts` type generated that also contains definitions.

Mapping nested classes

Java and C# both support the definition of nested classes. However, Java supports two types of nested classes: static and non-static. The following points clarify how it does this:

- Java static nested classes are the same as C# nested classes and are translated directly
- Non-static nested classes, also known as inner classes, are somewhat different; additional rules apply:
 - A reference to an instance of the containing type must be provided as a parameter in the constructor to the inner class.
 - In the case of inheriting from an inner class, the derived class must be nested within a type. This type inherits properties from the class that contains the base inner class, and the derived class must provide a constructor of the same type as the C# containing type.

Mapping the Runnable interface

Java provides the `java.lang.Runnable` interface with a single method, `run()`, in order to implement delegation. The Android platform makes use of this interface in a number of places such as `Activity.runOnUiThread()` and `View.post()`.

C# provides the `System.Action` delegate for a method with a void return and no parameters; thus, it maps very nicely to the `Runnable` interface. The Android bindings provide overloads that accept an `Action` parameter for all API members that accept a `Runnable` interface in the native API.

The `IRunnable` overloads were also left in place so that types that are returned from other API calls can be used.

Enumerations

In many places, the Android APIs uses `int` constants as parameters to specify processing options. To increase type safety, the Android bindings create enumerations to replace `int` constants when possible. The following example shows the use of the `ActivityFlags.NewTaskEnum` value rather than the native `FLAG_ACTIVITY_NEW_TASK` constant:

```
myIntent.SetFlags (ActivityFlags.NewTask);
```

Another great advantage of using an enum class is the enhanced support that you get with code completion in IDEs such as Xamarin Studio and Visual Studio.

Resources

Xamarin.Android generates a file named `Resource.Designer.cs` in the `Resources` folder of your project. This file contains constants for all of the resources referenced in your app and serves the same purpose as the `R.java` file generated for traditional Android apps.

Attributes for the ApplicationManifest.xml file

Android applications have a manifest file (`AndroidManifest.xml`) that tells the Android platform everything it needs to successfully run the application, including the following features:

- The minimum API level required by the application
- Hardware/software features used or required by the application
- Permissions required by the application
- The initial activity to start when the application is launched
- Libraries required by the application

Xamarin.Android provides a robust set of .NET attributes that can be used to adorn your C# classes so that much of the information required in `ApplicationManifest.xml` will be automatically generated at compile time. Use of these attributes simplifies the task of keeping the manifest in sync with your code. For example, if you rename an `Activity` class, the next time you compile, the corresponding `<Activity/>` element in the manifest is automatically updated.

The following example demonstrates the use of the `Activity` attribute to specify the launch activity for an app:

```
[Activity (Label = "My Accounts", MainLauncher = true)]
public class MyAccountsActivity : Activity
{
    ...
}
```

This will result in the following entry in the `ApplicationManifest.xml` file:

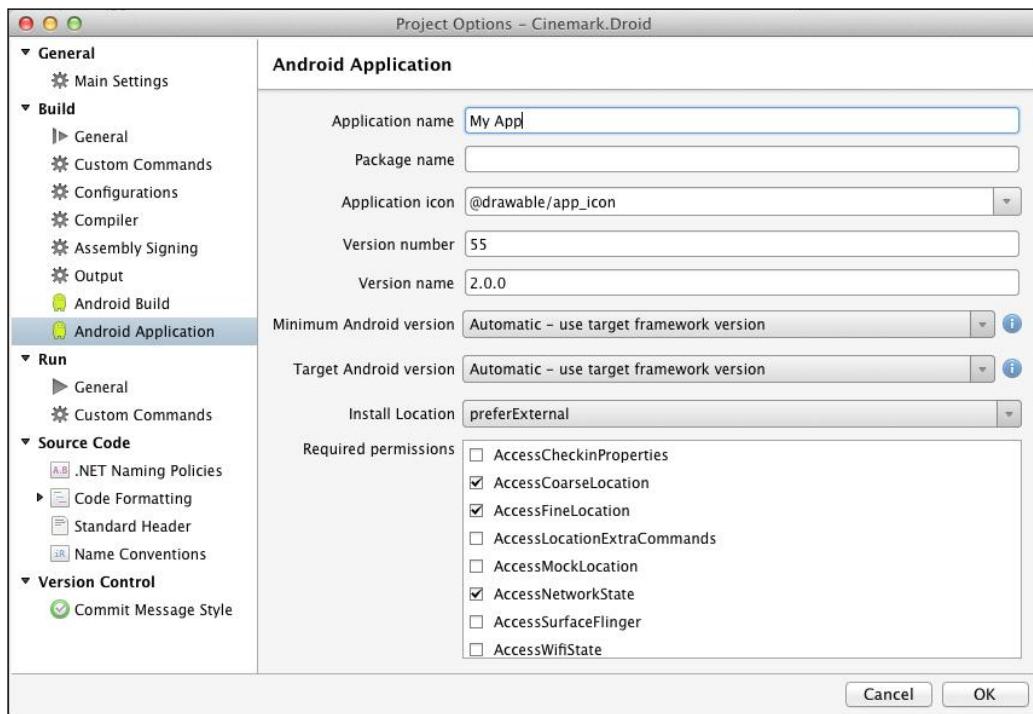
```
<activity android:label="My Accounts"
    android:name="myapp.MyActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
```

```
    android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
```

Although .NET attributes are a convenient way to keep your code and manifest file in sync, the use of these attributes is not required.

Editor for the ApplicationManifest.xml file

Xamarin Studio also provides an editor for `ApplicationManifest.xml`. This can be used instead of attributes or to edit content that cannot be set through the use of attributes such as the hardware/software features required and permissions. The following screenshot depicts the editor:



Garbage collection

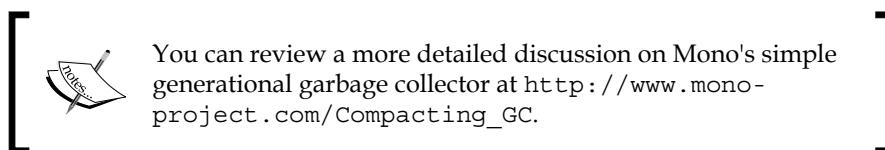
Since Xamarin.Android apps run in two different VMs, garbage collection is somewhat complex and creates some interesting challenges. Therefore, we have devoted significant time to discuss this process. Xamarin.Android uses Mono's simple generational garbage collector, which supports two types of collections called minor and major:

- **Minor collections:** These collections are cheap and thus invoked frequently. Minor collections collect recently allocated and dead objects and are invoked after every few MB of allocations. You can manually invoke a minor collection with the following code:

```
GC.Collect(0)
```

- **Major collections:** These collections are expensive and are thus invoked less frequently. Major collections reclaim all dead objects and are only invoked when memory is exhausted for the current heap size. You can manually invoke a major collection with the following code:

```
GC.Collect() or GC.Collect(GC.MaxGeneration).
```



Before we continue with the discussion, it will help us if we group objects in an app into the following categories:

- **Managed objects:** These are any C# objects you create from standard libraries such as the Mono runtime libraries. They are garbage collected like any other C# object and have no special connection to any classes from the Android bindings.
- **Java objects:** These are Java objects that reside in the Dalvik VM that were created as a part of some process, but not exposed to a managed object through JNI. These objects are collected as any other Java object, and there is little that we need to discuss about them.
- **Peer objects:** As we mentioned earlier, peer objects are a managed object and Java object pair that communicates via JNI. It works together to carry out the functionality of an Android app.

JNI global and weak references

JNI references come in a couple of different types and they have a big impact on when objects can be collected. Specifically, we will discuss two types of JNI references, which are global and weak references:

- **Global reference:** A JNI global reference is a reference from "native", or in our case managed code, to a Java object managed by the Dalvik VM. A JNI global reference is established between peer objects when they are initially created. A JNI global reference will prevent the Dalvik garbage collector from performing the required action as it indicates the object is still in use.
- **Weak reference:** A JNI weak reference also allows a managed object to reference a Java object, but the difference is that a weak reference will *not* prevent the Dalvik VM GC from collecting it.

We will see how these differ later on in this chapter.

Mono collections

Mono collections are where the fun happens. As mentioned earlier, simple managed objects are collected normally, but peer objects are collected by performing the following steps:

1. All managed peers are eligible for Mono collection, meaning they are not referenced by any other managed objects. They have their JNI global reference replaced with a JNI weak reference. This allows the Dalvik VM to collect the Java peer if no other Java objects in the VM reference them.
2. A Dalvik VM GC is invoked that allows the Java peers with weak global references to be collected.
3. Managed peers with a JNI weak reference, as created in step 1, are evaluated. If the Java peer has been collected, then the managed peer is also collected. If the Java peer has not been collected, then it is replaced with a JNI global reference and the managed peer is not collected until a future GC.

The end result is that an instance of a managed peer will live as long as it's referenced by a managed code or its corresponding Java peer is referenced by a Java code. To shorten the lifetime of peers, dispose of peer objects when they are no longer needed.

Best practice

Calling the `Dispose()` method manually severs the connection between the peers by freeing the JNI global reference, thus allowing each VM to collect the objects as soon as possible.

Automatic collections

From Xamarin.Android 4.1.0 version onwards, a full garbage collection is performed automatically when a gref threshold has crossed 90 percent of the known maximum gref values for the platform.

When you perform an automatic collection, a message similar to the following is displayed in the debug log:

```
I/monodroid-gc(PID): 46800 outstanding GREFs. Performing a full GC!
```

Invocations of the automatic GC are nondeterministic and might not happen at best times. If you are experiencing a pause in processing, look for messages in the logcat that might indicate that an automatic GC occurred. When this scenario occurs, you can consider when you can use `Dispose()` to reduce the lifetime of the peer.

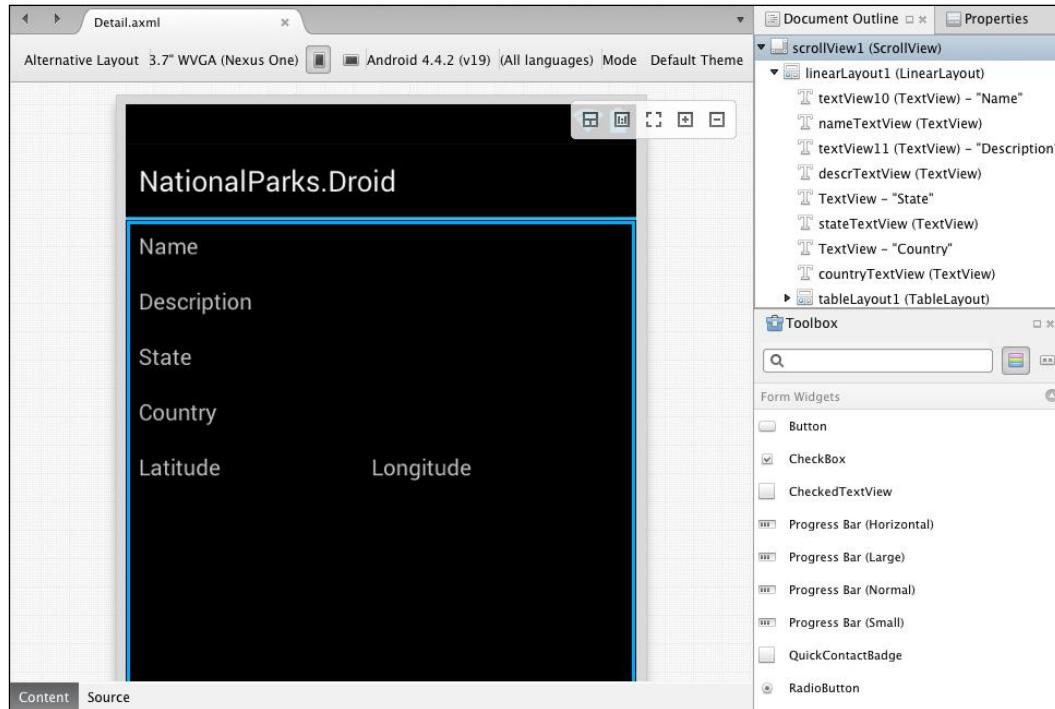
Helping the GC

There are a number of ways in which you can help the GC with the collection process. The following sections provide some additional thoughts:

- **Disposing of peer objects:** Get rid of managed peers when they are no longer needed and consider invoking a minor GC. As we mentioned earlier, the Mono GC does not have a complete picture of the memory situation. To Mono, peer objects appear to take up only 20 bytes because the MCWs don't add instance variables. Hence, all of the memory is associated with the corresponding Java object and allocated to the Dalvik VM. If you have an instance of `Android.Graphics.Bitmap` loaded with a 2 MB image, the Mono GC only sees the 20 bytes and thus disposing of the object will be a low priority to the GC.
- **Reduce direct references in peer objects:** Whenever a managed peer is scanned during GC, the entire object graph is scanned, meaning every object it directly references is scanned as well. Objects with a significant number of direct references can cause a pause when the GC runs.
- **Minor collections:** Minor collections are relatively cheap. You can consider invoking minor collections at the end of an activity or after completing a significant set of service calls or background processing.
- **Major collections:** Major collections are expensive and should rarely be performed manually. Only consider manually invoking a major collection after a significant processing cycle when a large amount of resource has been freed and you can live with a pause in the app's responsiveness.

Xamarin.Android Designer

Xamarin provides a plugin for Xamarin Studio that can be used to design layout files for Xamarin.Android apps. The designer supports a **Content** mode for visual drag-and-drop and a **Source** mode for XML-based editing. The following screenshot depicts the designer opened in the **Content** mode:



Summary

In this chapter, we reviewed the architecture of Xamarin.Android, discussed the design goals, and looked at some of the details of its implementation. We also looked at how memory management works with a Xamarin.Android app. In the next chapter, we will start developing a Xamarin.iOS app.

4

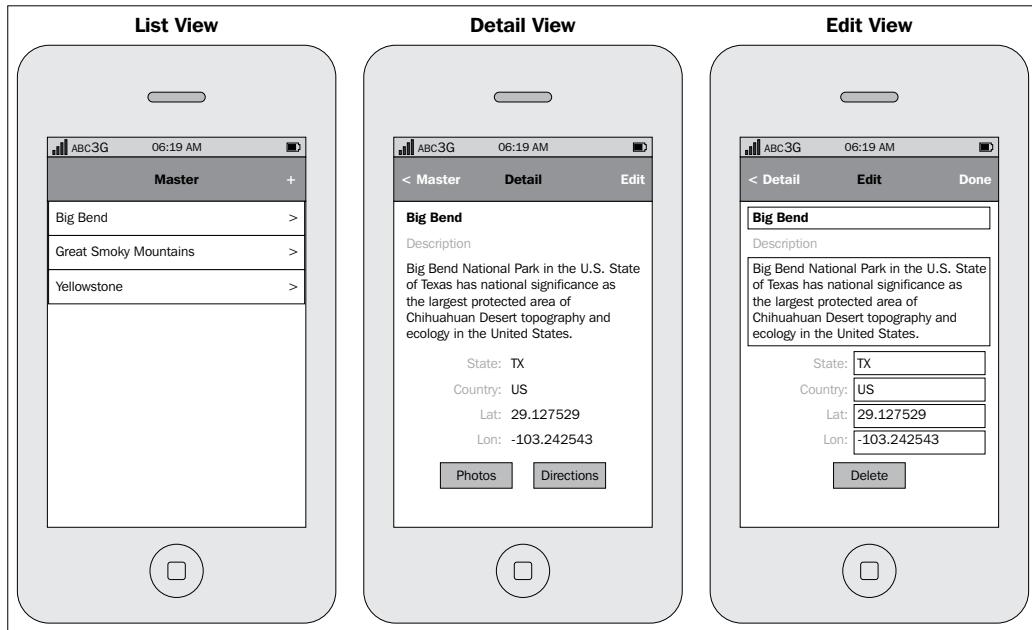
Developing Your First iOS App with Xamarin.iOS

In this chapter, we finally get to jump in and start writing some code. We will develop a sample app that demonstrates the basics of developing Xamarin.iOS apps and will cover the following topics:

- Overview of the sample app
- Creating a Xamarin.iOS app
- Running and debugging apps with Xamarin Studio
- Using Xamarin iOS Designer
- Extending the sample app
- MonoTouch.Dialog

The sample national parks app

In this chapter, we will create a sample app that we will continue to work with through *Chapter 8, Sharing with Xamarin.Forms*. The app will allow you to view, create, edit, and delete information about national parks, and will have a similar user interface and flow as the iOS 7 Contacts app. The following screen mock-ups depict how the user interface will be organized:



The following are the different views of national park apps:

- **List view:** This view displays a list of national parks that allows a park to be viewed and also a new park to be created
- **Detail view:** This view displays all the properties of a national park in read-only mode, and allows navigation to see photos of a park or see directions to a park
- **Edit view:** This view allows you to edit new or existing parks as well as delete parks

Creating the sample app

A Xamarin.iOS template will be used to create the sample app, giving us much of the required functionality that is already in place.

Throughout the chapter, we will present sample code from the downloaded solutions. Feel free to deviate in any manner to take the app in any direction you see fit.

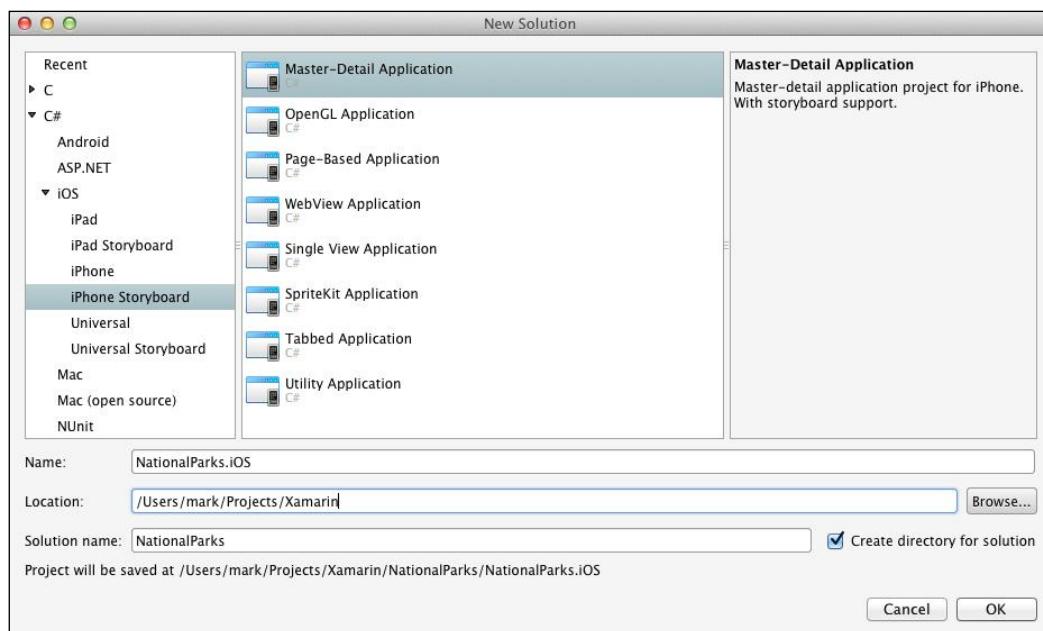
Downloading the example code



You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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.

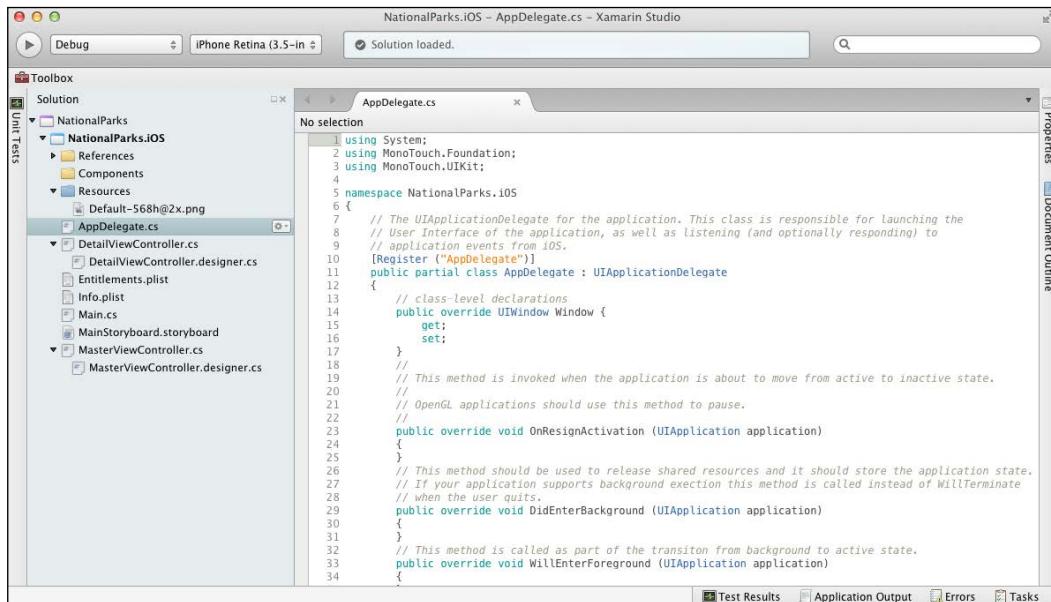
To create the national parks sample app, perform the following steps:

1. Launch Xamarin Studio.
2. From the **File** menu, navigate to **New | Solution**. The **New Solution** dialog box will be presented, as shown in the following screenshot:



3. Navigate to **C# | iOS | iPhone Storyboard** on the left-hand side of the dialog box and **Master-Detail Application** in the middle section.
4. Enter **NationalParks.iOS** in the **Name** field, select the location where you would like to place your code by clicking on the **Browse** button, change the **Solution name** to **NationalParks**, leave **Create directory for solution** checked, and click on **OK**.

5. One reason for naming the project `NationalParks.iOS` is that in future chapters, we will add a new project named `NationalParks.Droid` to the same solution. This project will clearly identify which platform is supported by each project.
6. Xamarin Studio will create both the solution and project folders, generate a number of files for solution and project, and then open the new solution and project. The following screenshot depicts Xamarin Studio with the newly created project open:



By selecting the Master-Detail template, Xamarin Studio has generated a functioning application with a master view (list) and a detail view along with everything that is needed to navigate between the two.

Let's take a brief look at what was automatically created:

- `MainStoryboard.storyboard`: A storyboard file containing the user interface definitions was created and named `MainStoryboard.storyboard`. Double-click on this file to open it in Xcode. You will notice that the storyboard contains two view controllers: `MasterViewController` and `DetailViewController` with a single segue between them.

- **MasterViewController:** The `MasterViewController.cs` and the corresponding `MasterViewController.designer.cs` files were created as a result of `MasterViewController` defined in the storyboard. `MasterViewController.cs` is the file where we will add code, while we override methods and add logic.
- **DataSource:** `MasterViewController` contains an inner class named `DataSource`, which is a specialization of `UITableViewSource`. The `DataSource` class is responsible for providing populated `UITableViewCell`s to the table view on `MasterViewController`.
- **DetailViewController:** The `DetailViewController.cs` and its corresponding `DetailViewController.designer.cs` files were created as a result of `DetailViewController` defined in the storyboard. This is used to display properties of a specific item from the table view on `MasterViewController`.

The Project Options view

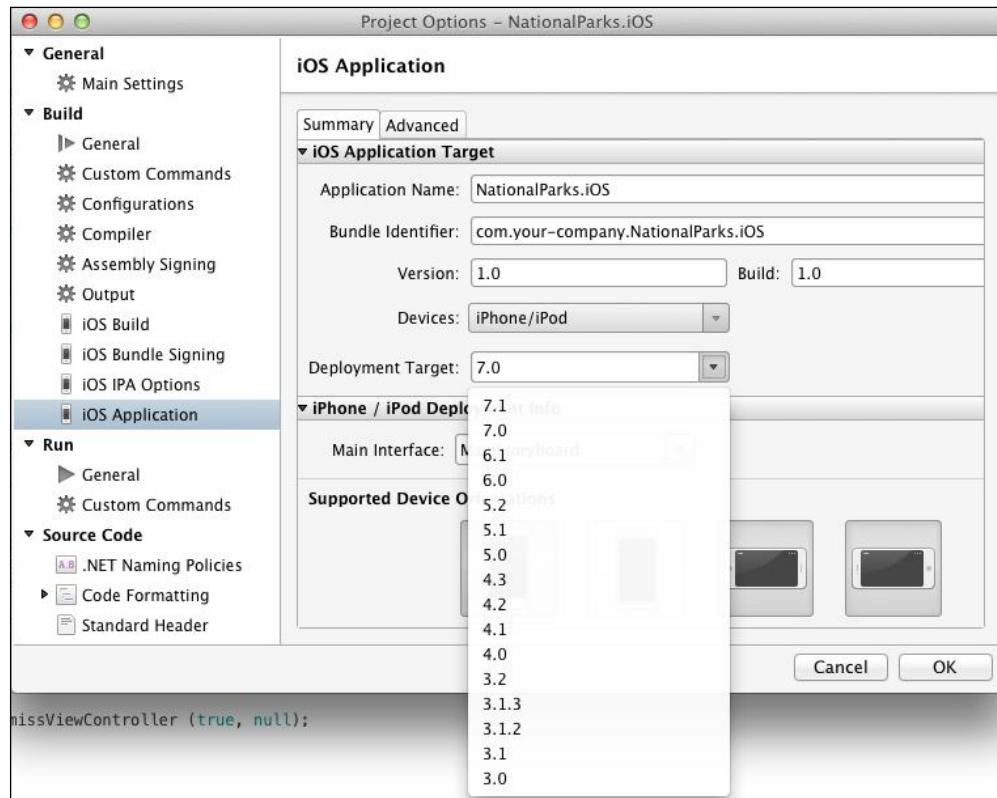
There are numerous options that can be set that affect how an iOS app is built and executed. These options can be viewed and adjusted from the **Project Options** view. The following sections are of the most interest for iOS apps:

- **iOS Application:** This includes settings that describe the application, including the devices supported, iOS target version, orientations supported, icons, and more
- **iOS IPA Options:** This includes settings related to creating an IPA package for ad hoc distribution
- **iOS Bundle Signing:** This includes settings that control how the bundle is signed during the build process
- **iOS Build:** This includes settings used by the compile and link process to optimize the resulting executable

Prior to running the app, we need to choose a setting for our target version of iOS. To adjust this setting, follow these steps:

1. Select the sample app project under the sample app solution in the **Solution** pad.
2. Right-click and select **Options**.

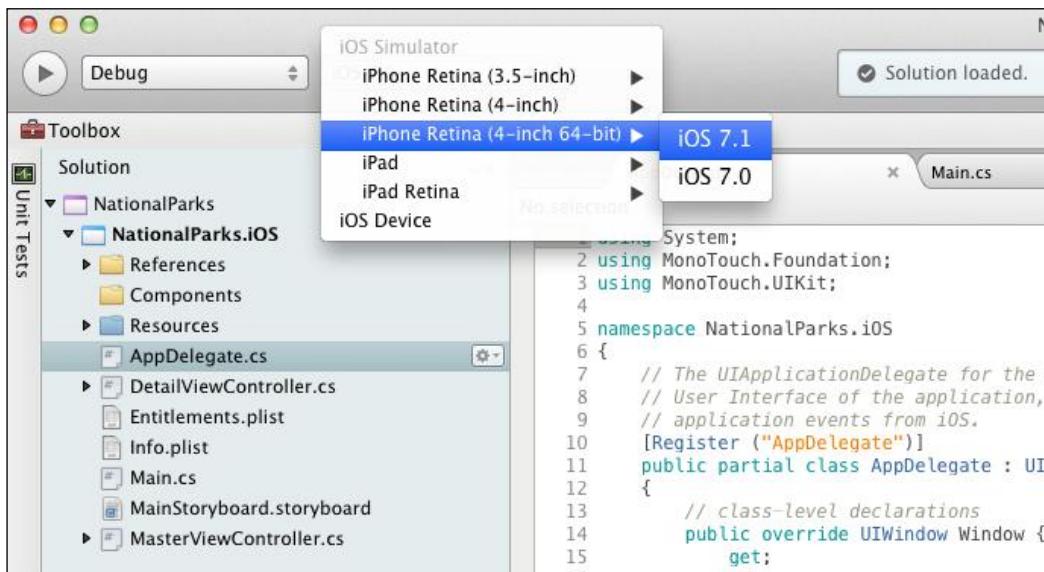
3. Select the **iOS Application** section and set the **Deployment Target** option to 7.0 and click on **OK**.



Running and debugging within Xamarin Studio

Now that we have a good understanding of what was created for us, let's take a few minutes to look at the capabilities Xamarin Studio provides to run and debug apps. The way in which a tool supports running and debugging apps has a big impact on developer productivity. Xamarin Studio provides a robust set of debugging capabilities comparable to the most modern development environments that can be used with either the iOS Simulator or a physical device. As with an iOS development, using a physical device provides the most accurate results.

The two dropdowns on the upper-left hand corner of Xamarin Studio control the type of build (**Release** or **Debug**) that will be produced and, when **Debug** is selected, which of the iOS Simulators should be used. Build types include **Ad-Hoc**, **AppStore**, **Debug**, and **Release**. All of these build types except **Debug** will be discussed in *Chapter 9, Preparing Xamarin.iOS Apps for Distribution*. The **Debug** build type is shown in the following screenshot:



Note that **Debug** is selected for the type of build and the various options available for the iOS Simulator. Selecting **iOS Device** allows you to debug the app on an attached device.

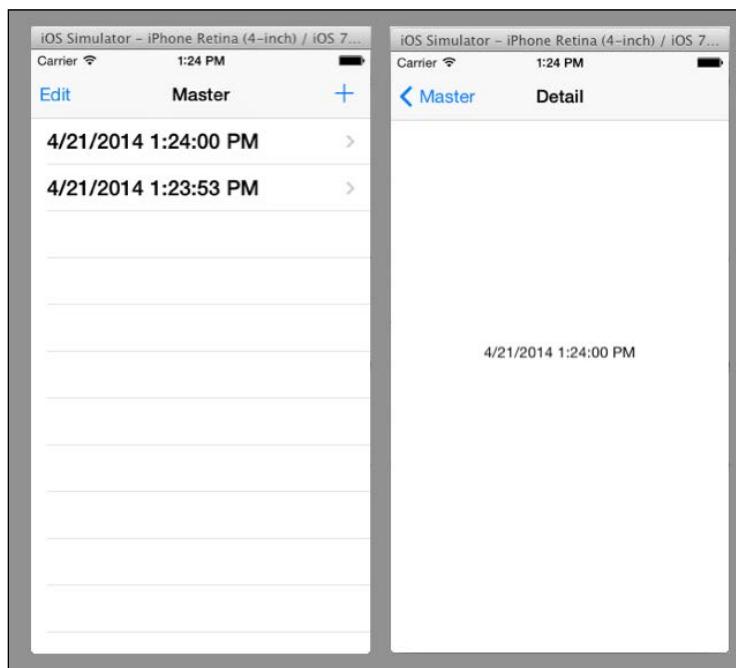
To debug an app, follow the given steps:

1. Select **Debug** for the build type, and select **iOS 7.1** from **iPhone Retina (4-inch 64-bit)** for the **iOS Simulator** option.
2. Start the debugging session by clicking on the **Start** button from the taskbar on the left-hand side. You can also initiate the debugging session by navigating to **Run | Start Debugging** from the main menu bar.

3. Xamarin Studio will compile the app, launch the iOS Simulator, install the app on the simulator and finally launch the app. Xamarin Studio keeps you informed of what is going in the status window in the middle of the taskbar. The following screenshot shows the status window during the build process:

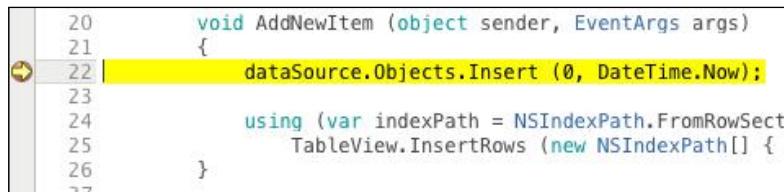


4. An empty list is initially presented. Click the + (add) button a few times and you will see the date/time fields being added to the list. Select an entry and the **Detail** view is displayed, as shown in the following screenshot:



5. Open `MasterViewController.cs` by double-clicking the **Solution** pad on the left-hand side. Set a breakpoint on the first line in the `AddNewItem()` method.

- Click on the + (add) button in the app. You will notice the app has stopped on the breakpoint, as follows:



```

20     void AddNewItem (object sender, EventArgs args)
21     {
22 |         dataSource.Objects.Insert (0, DateTime.Now);
23
24         using (var indexPath = NSIndexPath.FromRowSection (
25             TableView.InsertRows (new NSIndexPath[] {
26
27

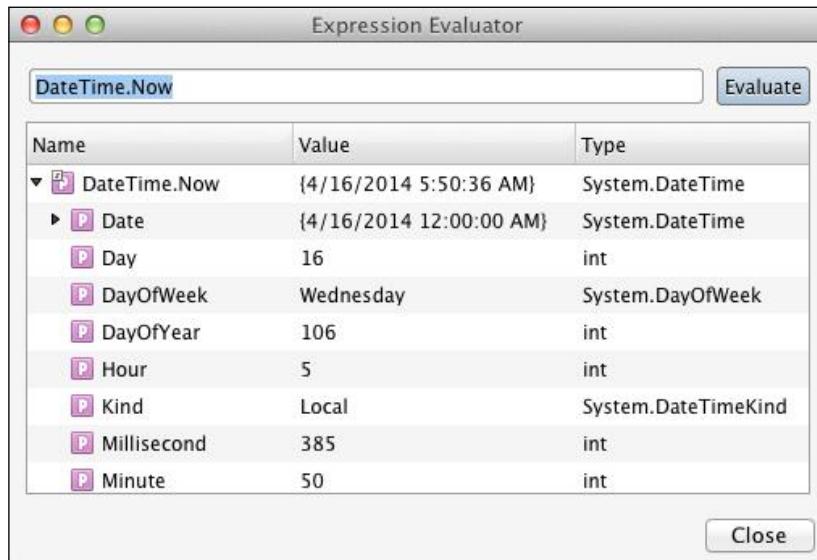
```

- You will find the basic flow controls in the taskbar. These allow you to continue execution, step over current line, step into current function, and step out of the current function. The taskbar will appear:

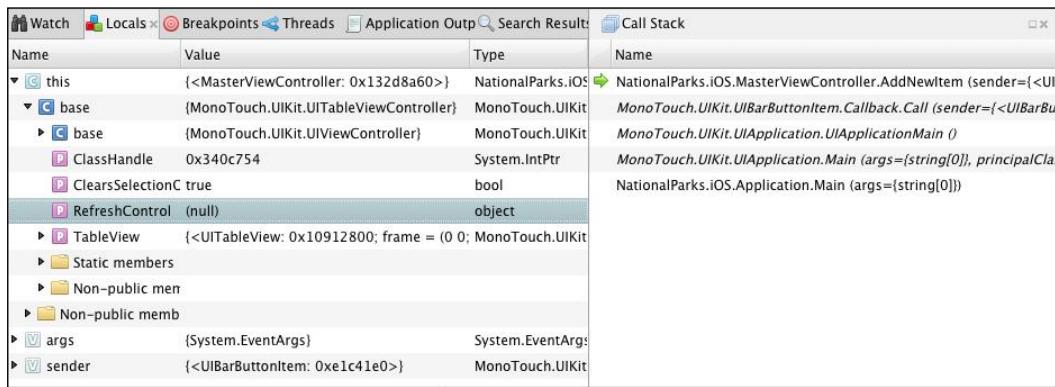


 A complete set of flow control and debugging options can be found under the **Run** main menu.

- From the first line in `AddNewItem()`, select `DateTime.Now`, right-click, and select **Expression Evaluator**. This dialog box allows you to quickly view the status of objects during your app's execution, as follows:



9. You will also notice a set of panels at the bottom of Xamarin Studio, which contain tabs for **Watch**, **Locals**, **Breakpoints**, **Threads**, **Application Output**, and **Call Stack**.



10. Click the continue icon to allow the app to continue running.

As you can see from the previous exercise, Xamarin Studio and the iOS Simulator provide a robust set of features to run and debug applications.

Extending the sample app

Now, it's time to extend the app. We have two primary tasks before us:

- Create a way for national parks to be loaded and saved from a file
- Enhance the user interface to show all of the appropriate attributes and allow to view and edit data

Storing and loading national parks

We will use a simple JSON-formatted text file to store information. .NET provides libraries to accomplish this, but the library I have had the most success with is Json.NET. Json.NET is an open source library created by James Newton-King, and this is definitely worth considering. Json.NET is also available in the Xamarin component store, so we can add it to our project directly from there.

Adding Json.NET

To add Json.NET to the sample app, perform the following steps:

1. Expand the `NationalParks.iOS` project, select the `Components` folder, and choose **Edit Components**.
2. In the upper-right corner, click on **Get More Components** and enter `Json.NET` in the search field.
3. Select `Json.NET` from the list and choose **Add to App**.

Creating an entity class

We now need an entity class that represents our subject: national parks. This will be a simple .NET class with a handful of properties.

To create an entity class, perform the following steps:

1. Right-click on `NationalParks.iOS` project and select the **New File** option.
2. In the **New File** dialog box, select the **General** section, select **Empty Class**, enter `NationalPark` in the **Name** field, and click on **New**.

The following code demonstrates what is needed for the entity class:

```
public class NationalPark
{
    public NationalPark ()
    {
        Id = Guid.NewGuid ().ToString ();
        Name = "New Park";
    }

    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public double? Latitude { get; set; }
    public double? Longitude { get; set; }

    public override string ToString ()
    {
        return Name;
    }
}
```

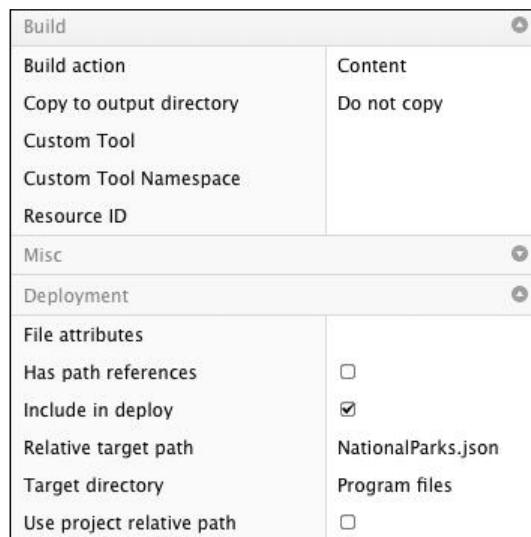
Adding a JSON-formatted file

Now, we need a file filled with JSON-formatted national parks. You can find such a file in the assets folder of the downloaded solutions. The name of the file is `NationalParks.json`.

To add the data file to the project, perform the following steps:

1. Copy the `NationalParks.json` file to the `NationalParks.iOS` project folder.
2. Right-click on the `NationalParks.iOS` project and choose **Add Files**, select `NationalParks.json` and click on **Open**.
3. Double-click on the `NationalParks.json` file to open it and view the content.

There are several file properties that must be set that determine how the file is handled during the compilation and deployment process. We want the file to be treated as content and placed in the same folder as the app during deployment. The following screenshot shows the settings required to accomplish this. The panels to adjust these settings are in the **Properties** tab on the right-hand side of Xamarin Studio.



This is not an ideal location for apps to store their data for a production distribution, but will work for our purpose in this chapter. In *Chapter 5, Developing Your First Android App with Xamarin.Android*, we will build a more robust storage mechanism when we discuss sharing code between iOS and Android apps.

Loading objects from a JSON-formatted file

Now, we need to add the logic to load data from the file to a list.

To load objects from a file, perform the following steps:

1. As you recall, when our app was generated, the `UITableViewSource` file placed in `MasterViewController.cs` used a `List<object>` object to populate the list. We need to convert this to `List<NationalPark>` `Parks`, as follows:

```
readonly List<NationalPark> parks;
```

Note that we do not instantiate the `Parks` list; Json.NET will do this for us when we deserialize the JSON string.

2. We also need to convert the `Objects` property defined on `DataSource` to the following:

```
public IList<NationalPark> Parks {get {return parks;}}
```

3. Add the using clauses for `System.IO` and `Newtonsoft.Json` in preparation to add the load and deserialize steps:

```
using System.IO;  
.  
.  
.  
using Newtonsoft.Json;
```

4. The JSON file will be placed in the app folder; the `Environment.CurrentDirectory` property gives us the path to this location. Loading objects from this file requires two basic steps. The first step is to read the text into a string with `File.ReadAllText()`. The second step is to deserialize the objects into a list using `JsonConvert.DeserializeObject<>()`. The following code sample demonstrates what needs to be placed in the constructor of the `DataSource` class:

```
string dataFolder = Environment.CurrentDirectory;  
string serializedParks =  
    File.ReadAllText (Path.Combine(dataFolder,  
        "NationalParks.json"));  
parks =  
    JsonConvert.DeserializeObject<List<NationalPark>>  
        (serializedParks);
```

Saving objects to a JSON-formatted file

Saving objects to a JSON-formatted file is just as simple as loading them. Simply call `JsonConvert.SerializeObject()` to create a JSON representation of the object(s) in a string and write the resulting string to a text file using `File.WriteAllText()`. The following code demonstrates what is needed:

```
string dataFolder = Environment.CurrentDirectory;
string serializedParks =
    JsonConvert.SerializeObject (dataSource.Parks);
File.WriteAllText(Path.Combine(dataFolder,
    "NationalParks.json"), serializedParks);
```

We will use this logic in the upcoming section titled *Implementing the Done Clicked event handler*.

Running the app

We are now ready to take a look at some of our work. Run the app and notice the following:

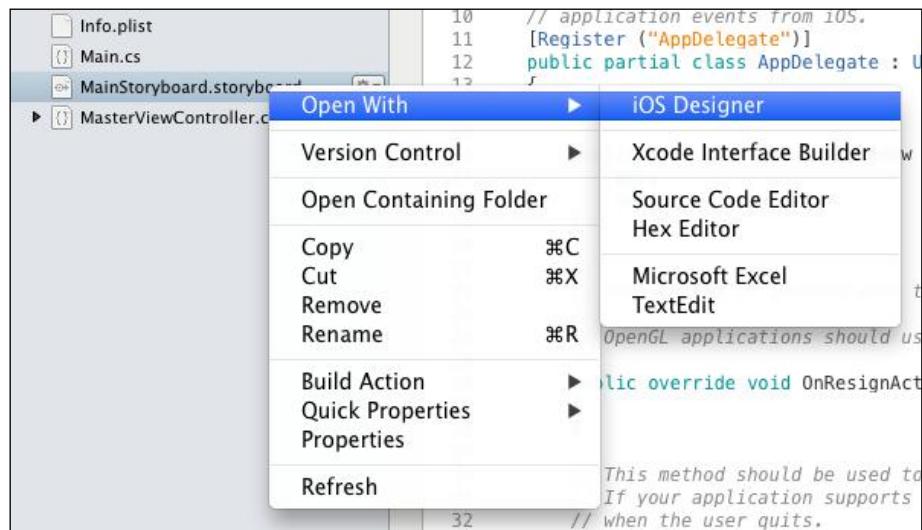
- `MasterViewController` is populated with information from `NationalParks.json`
- Selecting a park displays `DetailViewController` populated with the name of the park
- Clicking on the **Add** button from `MasterViewController` adds a new park with the name **New Park**

Enhancing the UI

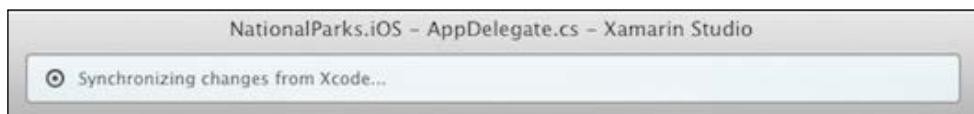
We will now turn our attention to create a more robust UI that will support listing items, view item details, and edit new and existing items. This is a common pattern for mobile apps and we already have about 75 percent of what we need. We need to make the following additions:

1. Add a new view controller named `EditViewController` that can be used to edit new or existing national parks.
2. Change the **Add** button on `MasterViewController` to open a new national park entry in `EditViewController`.
3. Add fields to `DetailViewController` that will display all the properties of a national park and an **Edit** button that will navigate to `EditViewController` to edit the current item.

As we discussed in *Chapter 2, Demystifying Xamarin.iOS*, we have two options to edit storyboards: Xcode Interface Builder and the Xamarin.iOS Designer. Either of these tools can be used based on preference. Within Xamarin Studio, you can choose which tool to launch by selecting a storyboard file, right-click on it and select **Open With**. The following screenshot shows the storyboard context menu:



The rest of this chapter will be based on using the Xamarin.iOS Designer. If you choose to work with Xcode Interface Builder, you need to be aware that when changes are made in Xcode, there is a synchronization process that takes place when the Xamarin Studio becomes active again. This process synchronizes changes made in Xcode with C# designer class files and creates appropriate outlets and actions. The following screenshot shows Xamarin Studio's status bar during synchronization:



If you would like more guidance with Xamarin.iOS Designer or need a kick start or refresher course on using Xcode Interface Builder, the following links provide tutorials:

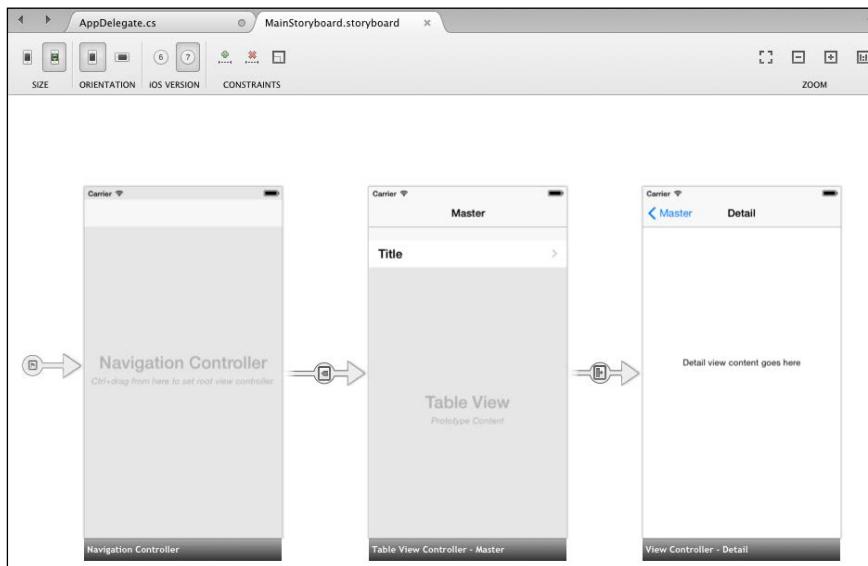


- Xamarin Tutorial for using Xamarin.iOS Designer available at http://developer.xamarin.com/guides/ios/user_interface/designer/
- Apple Tutorial for Xcode Interface Builder available at https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/chapters/edit_user_interface.html
- Xamarin Tutorial for using Xcode Interface Builder available at http://docs.xamarin.com/guides/ios/user_interface/tables/part_5_-_using_xcode,_interface_builder,_and_storyboards/

Touring the Xamarin.iOS Designer

Xamarin.iOS Designer provides a full set of tools to create and edit storyboard files. As this might be your first time using the tool, we will devote a few minutes to get familiar with it. To do so, follow the given steps:

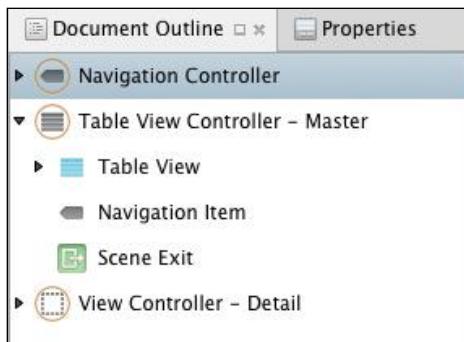
1. Double-click on `MainStoryboard.storyboard` to open the storyboard in Xamarin.iOS Designer. You will see `NavigationController`, `MasterViewController`, `DetailViewController`, and segues that connect everything, as shown in following screenshot:



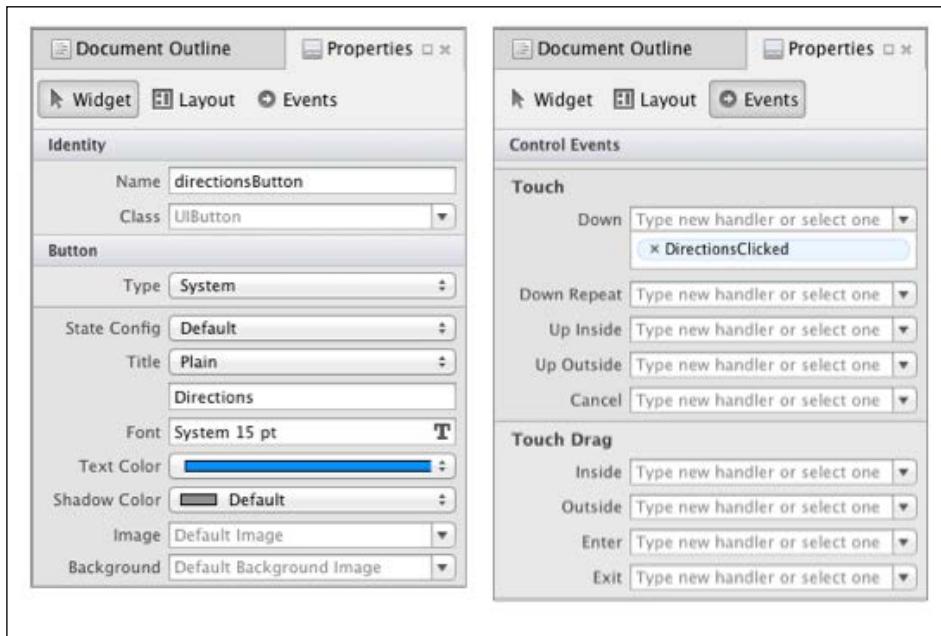
2. Note the **Toolbox** pad in the lower right-hand corner of Xamarin Studio. It contains all the items that can be used within a storyboard. You can search for items using the search field. The **Toolbox** pad is shown in the following screenshot:



3. Note the **Document Outline** pad in the upper-right hand corner of Xamarin Studio. This view depicts the content of the storyboard in a hierarchical form that you can use to drill down to view increasing levels of detail. The **Document Outline** pad is very helpful to view and select specific elements in a storyboard, as shown in the following screenshot:



4. Note the **Properties** pad in the upper-right hand corner of Xamarin Studio; you can access it by clicking on the tab labeled **Properties**. The **Properties** pad allows you to edit properties for the currently selected item. Entering a name for a control in the **Widget** section will automatically create an outlet and entering names in the **Events** section will automatically create an action. The **Properties** pad is shown in the following screenshot:



5. Note the top of the designer that contains a number of controls to adjust options, such as iOS version, device size, device orientation, and level of zoom. There are also controls to establish constraints, as shown in the following screenshot:



6. Selecting items in the designer can be a little tricky, particularly when selecting a view controller. If you click in the middle of the view controller, the view will be selected and not the view controller. There are three different ways to select a view controller:
 - Right-click in the middle of the view controller and navigate to **View Controller | Select**, as shown in the following screenshot:



- Click on the bar at the bottom of the view controller, as shown in the following screenshot:



- Select the view controller in the **Document Outline** pad

Adding EditViewController and segues

With a basic understanding of Xamarin.iOS Designer, we are now ready to add a new view controller and segues.

To add `EditViewController` and segues, perform the following steps:

1. Double-click on `MainStoryboard.storyboard` to open the storyboard in Xamarin.iOS Designer. You will see `MasterViewController` and `DetailViewController` in the file with a segue between them.
2. Create a new `UIViewController` by selecting the **View Controller** item from the **Toolbox** pad and drag-and-drop it on the designer view.

3. Name the new view controller `EditViewController` by clicking on the bar at the bottom to select it, switch to the **Properties** pad, and enter `EditViewController` for the **Class** field. The following screenshot depicts the **Properties** pad:



4. Add a Bar Button Item to the right-hand side of the Navigation Item on `DetailViewController` and set the **Identifier** button to **Edit** on the **Widget** section of the **Properties** pad.
5. Add a push segue from the **Edit** button on `DetailViewController` to the new controller, `EditViewController`. Press and hold the *Ctrl* key, click and hold the **Edit** button, drag it to the bar at the bottom of `EditViewController`, let go of the mouse, choose **Push** and enter `editFromDetail` for the **Identifier** option on the **Widget** section of the **Properties** pad.
6. Add a Bar Button Item to the right-hand side of the Navigation Item on `MasterViewController` and set the **Identifier** button to **Add**.
7. Add a push segue from the **Add** button on `MasterViewController` to the new controller, `EditViewController`. Press and hold the *Ctrl* key, click and hold the **Add** button, drag it to the bar at the bottom of `EditViewController`, let go of the mouse, choose **Push**, and enter `editFromMaster` for the **Identifier** option on the **Widget** section of the **Properties** pad.
8. Add a Bar Button Item to the right-hand side of the Navigation Item on `EditViewController` and set the **Identifier** option to **Done**. Name the button `DoneButton`. Naming the button will create an outlet that can later be used as a reference to assign a traditional .NET event handler.

9. Add a label, `UILabel`, to the center of `EditViewController`. This will be used temporarily to display the name of an item, while we test and debug the navigation of the app. Name this `UILabel` instance as `editDescriptionLabel`.
10. Add a `UIButton` instance to `EditViewController` and set the **Title** option to **Delete**. Add an action named `DeleteClicked` to the **Touch Down** event in the **Events** section of the **Properties** pad. Creating an action will generate a partial method that we can later complete with logic to implement the `DeleteClicked` event handler.
11. Save all of the changes made.
12. Now, we need to write some code to tie everything together. Let's start by looking at some of the code that was generated as a result of our work in Xamarin.iOS Designer. You will find two files that have been added for `EditViewController`, a designer file named `EditViewController.designer.cs` nested under a nondesigner file named `EditViewController.cs`. Double-click on the designer class to view the contents, as shown in the following code snippet:

```
[Outlet]
MonoTouch.UIKit.UINavigationItem DoneButton { get; set; }

[Outlet]

MonoTouch.UIKit UILabel editContent { get; set; }
[Action ("DeleteClicked")]
Partial void DeleteClicked (
    MonoTouch.Foundation.NSObject sender);
```



Note that `EditViewController` is a partial class; the two outlets and the actions were generated based on the specifications we made.



Implementing the `DoneClicked` event handler

For the **Done** button, we created an outlet so we will have a reference to the object that can be used to assign a .NET event handler at runtime. When **Done** is clicked on, we need to do a few things. First, check whether we are dealing with a new object and add it to the `_parks` collection. If so, then we need to save the `_parks` collection to `NationalParks.json`.

To implement the **Done** button, perform the following steps:

1. Create a method to save changes to `NationalParks.json`, as follows:

```
private void SaveParks()
{
    string dataFolder = Environment.CurrentDirectory;
    string serializedParks = JsonConvert.SerializeObject (_parks);
    File.WriteAllText(Path.Combine(dataFolder,
        "NationalParks.json"), serializedParks);
}
```

2. Create a .NET event handler named `DoneClicked` and add logic to add `_park` to the `_parks` collection. If it's a new park, call the `SaveParks()` method to save updates to `NationalParks.json`, and to return to the previous view controller, use the following code snippet:

```
private void DoneClicked (object sender, EventArgs e)
{
    if (! _parks.Contains (_park))
        _parks.Add (_park);

    SaveParks ();
    NavigationController.PopViewControllerAnimated (true);
}
```

Assign the `DoneClicked` event handler to the `Clicked` event on the `DoneButton` outlet in `ViewDidLoad()`.

```
public override void ViewDidLoad ()
{
    . . .     DoneButton.Clicked += DoneClicked;
}
```

Implementing the **DeleteClicked** action

We created an action for the **Delete** button, which caused a partial method to be created in the designer class. We now need to create an implementation for the partial method.

To implement the **Delete** action all you need to do is add a partial method implementation for `DeleteClicked` that removes `_park` from the `parks` collection and saves the change to the `NationalParks.json` file, which will then return to the `MasterViewController`. This can be done by:

```
partial void DeleteClicked (UIButton sender)
{
    if (_parks.Contains(_park))
        _parks.Remove(_park);

    SaveParks();

    NavigationController.PopToRootViewController(true);
}
```

The two approaches demonstrated to implement event handlers accomplish essentially the same thing without having a clear advantage over the other. As we don't have the event handler assignment in `ViewDidLoad()` for the action, it's slightly less coded. It really comes down to which method you prefer and become most comfortable with.

Passing data

All iOS apps have a need to navigate between views and pass data. As we are using storyboards and segues for the UI, most of the work related to navigation is done for us. However, we need to pass data between the views. There are two parts to accomplish this: define a way that a view controller will accept data and use this mechanism from the initiating view controller. As far as accepting data is concerned, this can be accomplished with the use of simple properties on the view controller, or by defining a method that accepts the data and saves it to private variables. We will go with defining a method to accept navigation data, which is also the approach the code that was generated for us uses.

To complete the logic to accept navigation data, perform the following steps:

1. Open `DetailViewController` and locate the `SetDetailItem` method.
2. Let's start by changing the name to be a little more meaningful. Select the `SetDetailItem` text in the editor, right-click and navigate to **Refactor | Rename**. Enter `SetNavData` and click on **OK**.

3. Let's also rename `ConfigureView ()` to `ToUI ()` using the same method.
4. Change the `SetNavData ()` method so that it accepts a list of `NationalPark` items as well as the single park that should be displayed and saves these parameters to a set of private variables. Also, remove the call to `ToUI ()`; we will move this to a more appropriate place in the next step, as shown in the following code:

```
IList<NationalPark> _parks;  
NationalPark _park;  
. . .  
public void SetNavData(  
    IList<NationalPark> parks, NationalPark park)  
{  
    _parks = parks;  
    _park = park;  
}
```

5. Override `ViewWillAppear ()` to call `ToUI ()`, as follows:

```
public override void ViewWillAppear (bool animated)  
{  
    ToUI ();  
}
```

6. Update `ToUI ()` so that it populates `UILabel` using the private `_park` variable, as follows:

```
void ToUI()  
{  
    // Update the user interface for the detail item  
    if (_park != null)  
        detailDescriptionLabel.Text = _park.ToString ();  
}
```

7. Now, add `SetNavData ()` and `ToUI ()` methods to `EditViewController` that has the same function as `DetailViewController`.

Now that we have taken care of receiving navigation data, we turn our attention to passing data. When using segues, iOS view controller has the `PrepareForSegue ()` method that can be overridden to prepare the target view controller for display. We need to override `PrepareForSegue ()` in both `MasterViewController` and `DetailViewController`.

To complete the logic to pass navigation data, perform the following steps:

1. Open `MasterViewController` and locate the existing `PrepareForSegue()` method.
2. `MasterViewController` actually has two segues: the original segue that navigates to `DetailViewController` and the new one we added that navigates to `EditViewController`. The `PrepareForSegue()` method provides a segue parameter that has an `Identifier` property that can be used to determine which navigation path is being taken. Change the code in `PrepareForSegue()` so that it calls `SetNavData()` on the appropriate view controller based on the segue identifier, as follows:

```
public override void PrepareForSegue (
    UIStoryboardSegue segue, NSObject sender)
{
    if (segue.Identifier == "showDetail") {
        var indexPath = TableViewIndexPathForSelectedRow;
        var item = dataSource.Parks[indexPath.Row];
        ((DetailViewController)segue.
            DestinationViewController).SetNavData
            (dataSource.Parks, item);
    }
    else if (segue.Identifier == "editFromMaster") {
        ((EditViewController)segue.
            DestinationViewController).SetNavData
            (dataSource.Parks, new NationalPark());
    }
}
```

3. Now, open `DetailViewController` and create an override for `PrepareForSegue()` that passes navigation data to `EditViewController`, as follows:

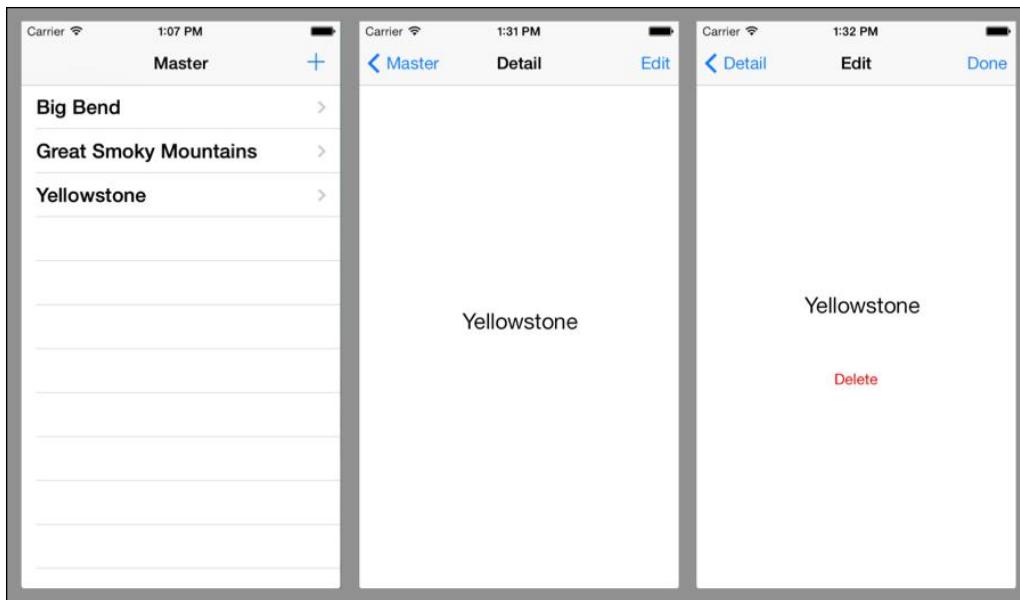
```
public override void PrepareForSegue (
    UIStoryboardSegue segue, NSObject sender)
{
    if (segue.Identifier == "editFromDetail") {
        ((EditViewController)segue.
            DestinationViewController).SetNavData
            (_parks, _park);
    }
}
```

Running the app

We have made a lot of changes and are now ready to run the app to test the basic navigation. Start the app and test navigation to the various views; observe the following:

1. When you click the **+** (add) button on `MasterViewController`, a new national park is displayed in `EditViewController`.
2. When you click the **Edit** button on `DetailViewController`, the app navigates to `EditViewController` that shows the current park.
3. When you click on **Done** on `EditViewController`, it will take you to the previous view controller, either `MasterViewController` or `DetailViewController`.
4. When you click on **Delete** on `EditViewController`, it will take you to `MasterViewController`.

The following screenshots depict what you should see:



Finishing the sample app

The view controllers and navigation are now in place. All we need now is to add some addition controls to view and edit information and a little logic.

Finishing DetailViewController

To finish `DetailViewController`, we need a set of `UILabel` controls that can be used to display the properties of a park, and add buttons that can initiate actions to view photos or receive directions.

To finish `DetailViewController`, perform the following steps:

1. Add a `UIScrollView` onto the View for `DetailViewController`.
2. Add `UILabel` controls for each property defined on `NationalPark` except for the `Id` property. Also add `UILabel` controls that can be used as labels for the properties. Use the screen mockups from the *The sample national parks app* section as a guide to lay out the controls.
3. Enter a name for each `UILabel` control that will be used to display park properties so that outlets can be created.
4. Update the `ToUI()` method so that the `UILabel` controls are populated with data from the park, as follows:

```
void ToUI()
{
    // Update the user interface for the detail item
    if (IsViewLoaded && _park != null) {
        nameLabel.Text = _park.Name;
        descriptionLabel.Text = _park.Description;
        stateLabel.Text = _park.State;
        countryLabel.Text = _park.Country;
        latitudeLabel.Text = _park.Latitude.ToString ();
        longitudeLabel.Text = _park.Longitude.ToString ();
    }
}
```

5. Add a `UIButton` instance with a title of photos with an action named `PhotoClicked` in the `Touch Down` event.
6. Add an implementation for the `PhotoClicked` action, which opens a URL to view photos on `www.bing.com` that uses the park's name as the search parameter:

```
partial void PhotosClicked (UIButton sender)
{
    string encodedUriString =
        Uri.EscapeUriString(String.Format(
            "http://www.bing.com/images/search?q={0}", _park.Name));
    NSUrl url = new NSUrl(encodedUriString);
    UIApplication.SharedApplication.OpenUrl (url);
}
```

7. Add a `UIButton` instance with a title of directions with an action named `DirectionsClicked` in the `Touch Down` event.
8. Add an implementation for the `DirectionsClicked` action, which opens a URL to receive directions to a park that uses the park's latitude and longitude coordinates:

```
partial void DirectionsClicked (UIButton sender)
{
    if ((_park.Latitude.HasValue) && (_park.Longitude.HasValue))
    {
        NSUrl url = new NSUrl (
            String.Format(
                "http://maps.apple.com/maps?daddr={0},{1}",
                _park.Latitude, _park.Longitude));

        UIApplication.SharedApplication.OpenUrl (url);
    }
}
```

9. Add appropriate constraints to `UIScrollView` and `UILabels` so that scrolling and layout works as desired in the landscape and portrait modes. Take a look at the example for more clarity.

Finishing EditViewController

To finish `EditViewController`, we need to add labels and edit controls in order to edit the park data. We also need to do some data conversion and save the updates.

To finish `EditViewController`, perform the following steps:

1. Add a `UIScrollView` instance on the View for `EditViewController`.
2. Add controls to the `EditViewController` class along with the corresponding outlets to allow editing of each property on the `NationalPark` entity. The `UITextField` controls can be used for everything except the `description` property, which is better suited to a `UITextView` control. Also add `UITextLabel` controls to label properties of the park. You can again use the screen mockups from the *The sample national parks app* section as a guide.
3. Update the `ToUI()` method to account for the new fields:

```
private void ToUI ()
{
    // Update the user interface for the detail item
    if (IsViewLoaded && _park != null) {
        nameTextField.Text = _park.Name;
```

```
        descriptionTextView.Text = _park.Description;
        stateTextField.Text = _park.State;
        countryTextField.Text = _park.State;
        latitudeTextField.Text = _park.Latitude.ToString();
        longitudeTextField.Text =
            _park.Longitude.ToString();
    }
}
```

4. Create a new method that moves data from the UI controls to the entity class prior to saving it, as follows:

```
void ToPark()
{
    _park.Name = nameTextField.Text;
    _park.Description = descriptionTextView.Text;
    _park.State = stateTextField.Text;
    _park.Country = countryTextField.Text;

    if (String.IsNullOrEmpty (latitudeTextField.Text))
        _park.Latitude =
            Double.Parse (latitudeTextField.Text);
    else
        _park.Latitude = null;

    if (String.IsNullOrEmpty (longitudeTextField.Text))
        _park.Longitude =
            Double.Parse (longitudeTextField.Text);
    else
        _park.Longitude = null;
}
```

5. Update the DoneClicked() action to call ToPark() in order to move values to the park object prior to saving changes to NationalParks.json:

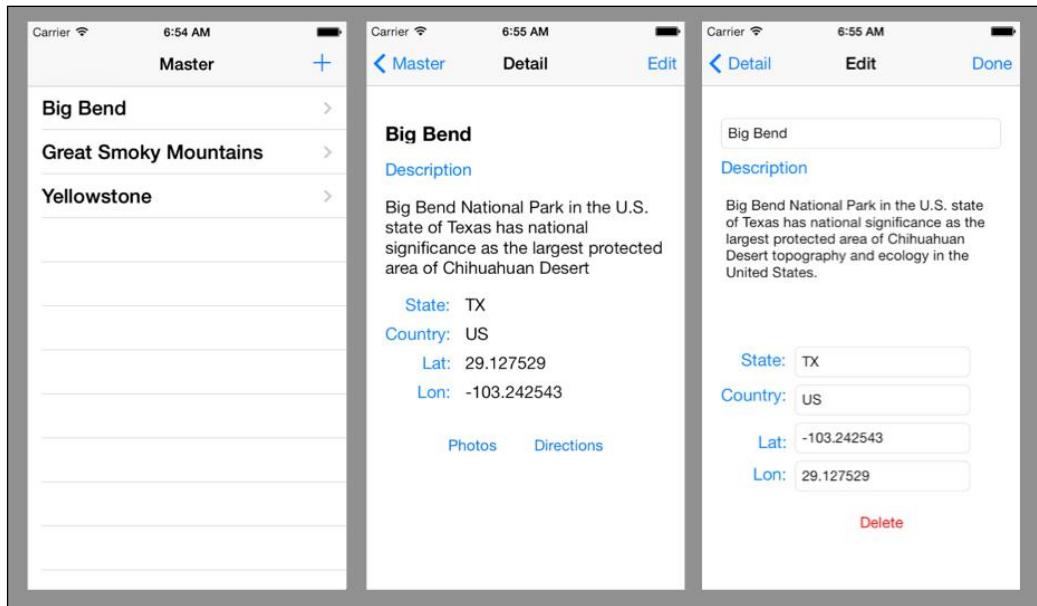
```
partial void DoneClicked (NSObject sender)
{
    ToPark ();
    . . .
}
```

6. Add appropriate constraints to UIScrollView and UITextFields so that scrolling and layout works as desired in landscape and portrait modes. Take a look at the reference solution for more clarity.

7. Add logic to scroll the active `UITextField` into view when the keyboard is displayed. There are several methods of accomplishing this. Refer to the example for reference solution.

Running the app

Okay, we now have a fairly functional app. Run the app in the simulator and test each screen and navigation path. The following screenshots show the final result of the three view controllers:



MonoTouch.Dialog

MonoTouch.Dialog (MT.D) is a framework for Xamarin.iOS that provides a declarative approach to develop the user interface and eliminates writing a lot of the tedious code. MT.D is based on using `UITableView` controls to provide navigation and allow users to interact with data.

More information about MT.D can be found at http://docs.xamarin.com/guides/ios/user_interface/monotouch.dialog/.

Summary

In this chapter, we created a sample Xamarin.iOS app and demonstrated the concepts that need to be understood to work with the Xamarin.iOS platform. While we did not demonstrate all of the features that we can use in an iOS app, you should now feel comfortable with how to access these features.

In the next chapter, we will build the same sample app for Android.

5

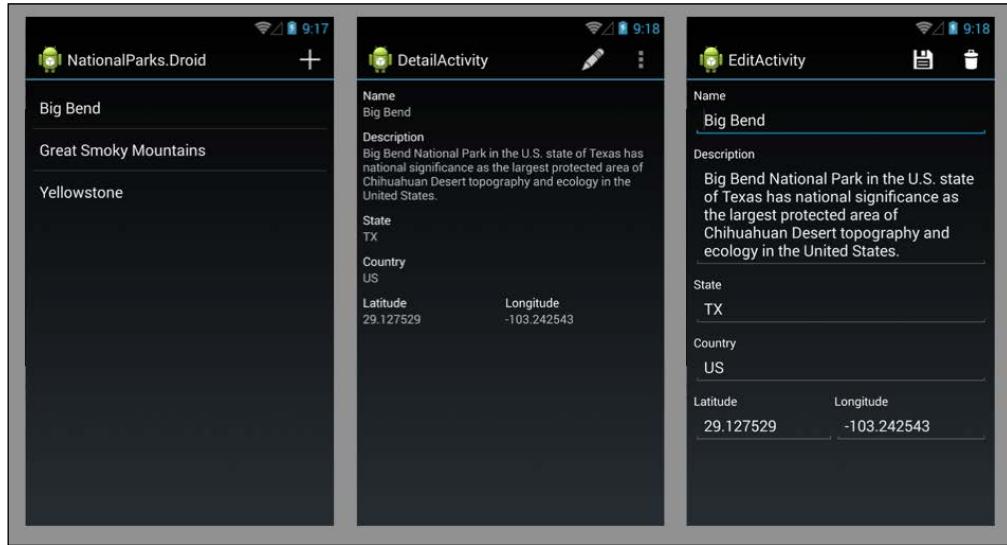
Developing Your First Android App with Xamarin.Android

In this chapter, we will develop a sample app similar to `NationalParks.iOS` from the last chapter using `Xamarin.Android`. This chapter covers the following topics:

- Overview of the sample app
- Creating a `Xamarin.Android` app
- Editing Android layout files with `Xamarin Studio`
- Running and debugging apps with `Xamarin Studio`
- Running and debugging apps with `Visual Studio`
- Inspecting compile-time generated elements of a `Xamarin.Android` app

The sample app

The sample app we will create in this chapter follows the same basic design as the `NationalParks.iOS` app from the previous chapter. To review the screen mockups and general description you can refer to the *The sample national parks app* section in *Chapter 4, Developing Your First iOS App with Xamarin.iOS*. The following screenshots show the Android screens from the provided solution:



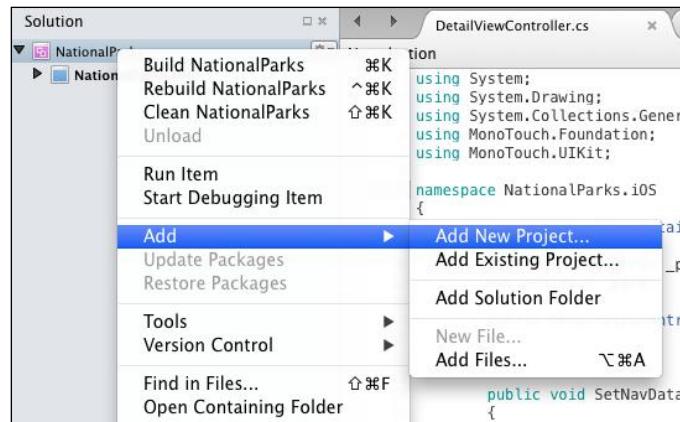
There is one design change we will introduce in the Android version of the app; we will create a singleton class to help manage loading and saving parks to a JSON-formatted file. This will be discussed further when we start building the singleton class in an upcoming section, *Creating the NationalParksData singleton*.

Creating NationalParks.Droid

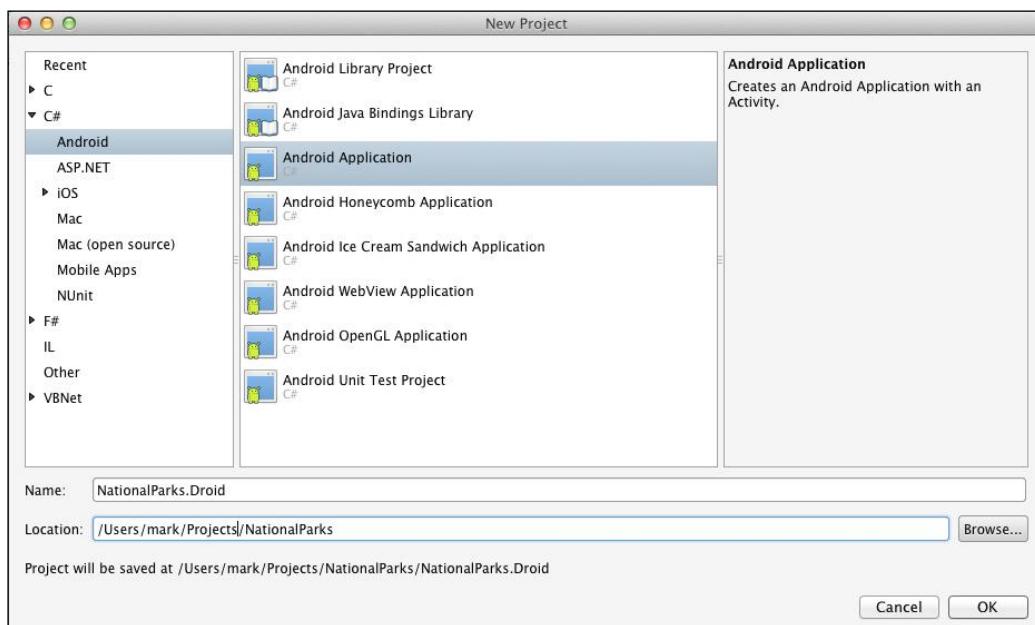
We will begin by creating a new Android project and adding it to our existing `NationalParks` solution created in the previous chapter. Xamarin Studio allows both Android and iOS projects to be part of the same solution. This proves to be very useful particularly as we move towards the later chapters that are focused on code reuse. You will find that the next chapter, *Chapter 6, The Sharing Game*, will show you exactly how to do this.

To create the national parks Android app, perform the following steps:

1. You first need to launch Xamarin Studio and open the NationalParks solution created in the previous chapter.
2. Following this, select the NationalParks solution in the **Solution** pad on the left-hand side of Xamarin Studio, right-click on it, and navigate to **Add | Add New Project...**, as shown in the following screenshot:



3. Navigate to **C# | Android** on the left-hand side of the dialog box and **Android Application** in the middle section, as follows:



4. You now need to enter `NationalParks.Droid` in the **Name** field and click on **OK**. Xamarin Studio will then create a new project, add it to the `NationalParks` solution, and open it.

Reviewing the app

A simple working app has been created that contains a number of files; let's take a few minutes to review what was created.

Resources

The Resources folder corresponds to the `res` folder in a traditional Java Android app. It contains subfolders with the various types of resources that can be used, including layouts, menus, drawables, strings, and styles. Subfolders within Resources follow the same naming conventions as in traditional Java Android apps, drawables, layouts, values, and so on.

The `Resource.designer.cs` file

`Resource.designer.cs` is a C# source file in the Resources folder that is generated by Xamarin.Android and contains constant ID definitions for all the resources in the app; it corresponds to the `R.java` file generated for Java Android apps.

The `MainActivity.cs` file

`MainActivity.cs` is a C# source file in the root of the `NationalParks.Droid` project and is the only activity added to the project. Open the file to view the contents. Note the attributes at the top of the class:

```
[Activity (Label = "NationalParks.Droid",
MainLauncher = true)]
```

The specification of `Label` and `MainLauncher` will affect the contents of `ApplicationManifest.xml`. Note the overridden `OnCreate()` method in the following code snippet:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);
```

```
// Set our view from the "main" layout resource
SetContentView (Resource.Layout.Main);

// Get our button from the layout resource,
// and attach an event to it
Button button =
    FindViewById<Button> (Resource.Id.myButton);

button.Click += delegate {
    button.Text = string.Format (
        "{0} clicks!", count++);
};

}
```

With the exception of the fact that `onCreate()` is using C# syntax, the code within it looks very similar to what you might find in a Java Android app. Near the top, the content is set to the Main layout file; `Resource.Layout.Main` is a constant defined in `Resource.designer.cs`. A reference to a `Button` instance is obtained by a call to `FindViewById()`, and then an event handler is assigned to handle click events.

The Main.axml file

`Main.axml` is an XML layout file located in the `Resources/layout` folder. Xamarin.Android uses the extension `.axml` for layout files rather than simply using `.xml`. Other than using a different extension, Xamarin.Android treats layout files in essentially the same way that a Java Android app does. Open `Main.axml` to view the contents; at the bottom of the screen, there are tabs to switch between a visual, content view, and a source or XML view. Notice that there is a single `Button` instance defined with `LinearLayout`.

Xamarin.Android honors the Android naming convention for layout folders, as follows:

- `Resources/layout`: This naming convention is used for a normal screen size (default)
- `Resources/layout-small`: This naming convention is used for small screens
- `Resources/layout-large`: This naming convention is used for large screens
- `Resources/layout-land`: This naming convention is used for a normal screen size in landscape mode

Project Options

There are many options that can be set that affect the way your app is compiled, linked, and executed. These options can be viewed and modified from the **Project Options** dialog box. The sections that are most interesting for Android apps are as follows:

- **Build | General:** This setting is used for the Target framework version
- **Build | Android Build:** This setting is used by the compile and link process to optimize the resulting executable
- **Build | Android Application:** This setting gives the default package name, app version number, and app permissions

To set the Target framework version for `NationalParks.Droid`, perform the following steps:

1. Select the `NationalParks.Droid` project in the **Solution** pad.
2. Right-click and select **Options**.
3. Navigate to **Build | General** and set the **Target Framework** option to **4.0.3 (Ice Cream Sandwich)** and click on **OK**.

Xamarin Studio Preferences

Xamarin Studio provides a **Preferences** dialog box that allows you to adjust various preferences that control how the environment operates. These are as follows:

- **Projects | SDK Locations | Android:** Using this option, you can control the location for the Android SDK, Java SDK, and Android NDK that should be used to compile and run apps
- **Projects | Android:** These settings affect how the Android Emulator is launched including command-line arguments

Running and debugging with Xamarin Studio

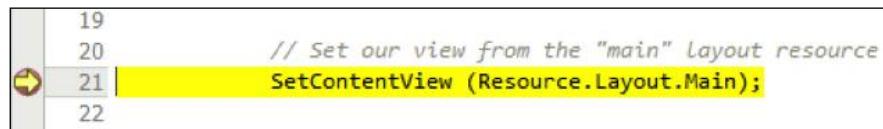
While the app we currently have is very simple, it is runnable, and now is a good time to take a look at how you can execute and debug `Xamarin.Android` apps. Apps can be executed in a number of ways; the two most common ways are the Android Emulator and a physical device.

Running apps with the Android Emulator

Xamarin.Android works with the Android Emulator to support executing and debugging your app. When Xamarin.Android is installed, a number of **Android Virtual Devices (AVD)** are automatically set up for your use. You can launch the AVD Manager from the **Tools** menu by choosing **Open Android Emulator Manager**.

To run `NationalParks.Droid`, perform the following steps:

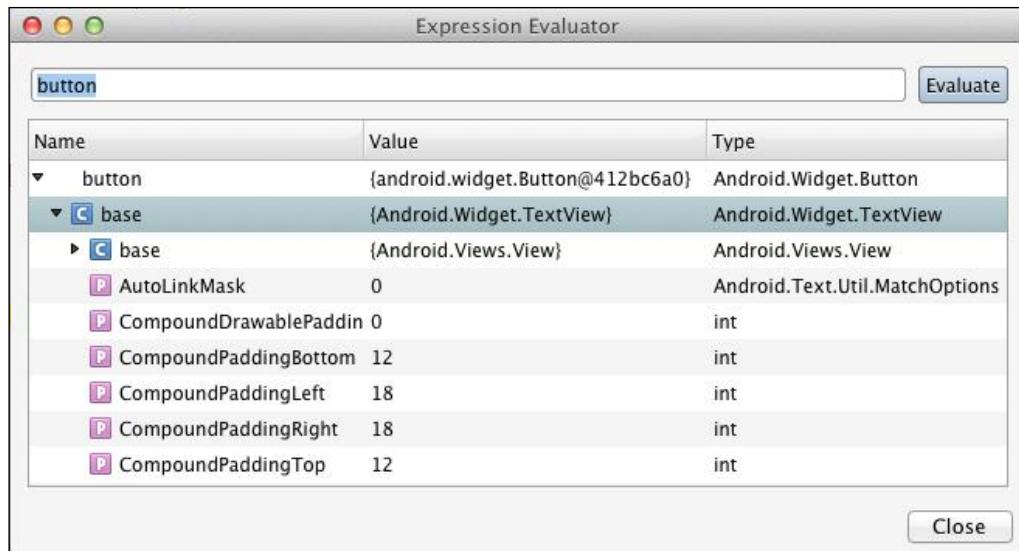
1. Click on the **Start/Stop** button on the left-hand side of the taskbar. You can also run the app by pressing **F5** or by navigating to **Run | Start Debugging**.
2. Then, select an AVD on the **Select Device** dialog box and click on **Start Emulator**.
3. When the emulator has completed startup, select the name of the running emulator instance in the **Devices** list and click on **OK**.
4. You now need to click on the **Hello World** button on the app and note the caption change.
5. Switch back to Xamarin Studio and stop the app by clicking on the **Start/Stop** button on the left-hand side of the taskbar.
6. Open `MainActivity.cs` and set a breakpoint on the `SetContentView()` statement in `OnCreate()` by clicking in the far-left margin of the editor window, which you can see in the following screenshot. At this point, restart `NationalParks.Droid`; the app will stop at the breakpoint:



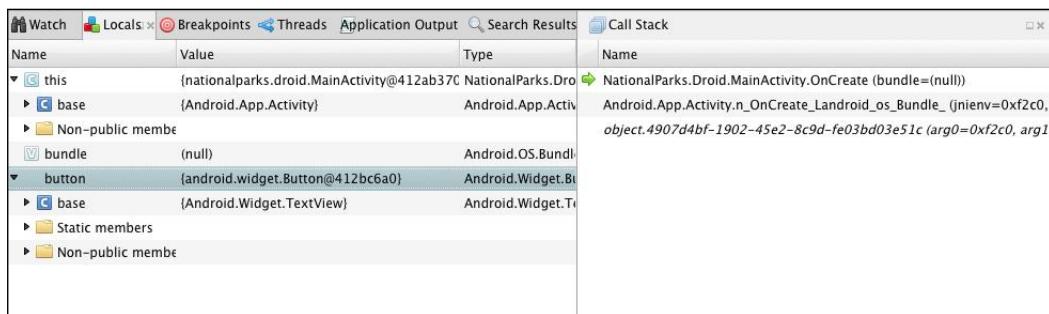
7. You will find the basic flow controls to step through execution in the taskbar. These allow you to (icons from left to right) continue execution, step over the current line, step into the current function, and step out of the current function:



8. Use the step controls to go to line 27, highlight the button in the text, right-click on it and select **Expression Evaluator**. The **Expression Evaluator** dialog box can be used to view the state of objects during the program execution, as follows:



9. You will also notice a set of panels at the bottom of Xamarin Studio that contains tabs for **Watch**, **Locals**, **Breakpoints**, **Threads**, **Application Output**, and **Call Stack**, as follows:



10. Click on the **Continue** button to allow the app to continue running.

As you can see, Xamarin Studio, in combination with the Android Emulator, provides a robust environment to execute and debug apps with most of the features that can be found in most modern IDEs.

You can make any modifications to the list of AVDs from the AVD Manager (**Tools | Open Android Emulator Manager**), and you can make any adjustments to the Android SDK from Android SDK Manager (**Tools | Open Android SDK Manager**).

Running apps on a physical device

Xamarin Studio also supports debugging apps running on a physical device. This is generally the most productive way to develop and debug apps as many of the device features can be challenging to configure and use in the emulator. There is really nothing special about getting Xamarin Studio to work with a device; simply go through the normal steps of enabling USB debugging on the device, attach the device to your computer, and start the app from Xamarin Studio; the device will show up in Xamarin Studio's **Select Device** dialog box. As you might be aware, on Windows, a special USB driver is required that corresponds with the device being used; generally OS X users are good to go.

The issues related to debugging with an emulator or physical device is not unique or even different because of the use of Xamarin; it's an issue that all Android developers face.

Running apps with Genymotion

A while ago, I became aware of another option to run Android apps. Genymotion is a product that is based on the VirtualBox virtualization platform. Genymotion provides a set of virtual device templates for many of the Android devices available on the market today. Once a virtual device is created, you simply start it and it will be selectable from Xamarin Studio's **Select Device** dialog box just like a running AVD.

With all the different device templates that come with Genymotion, it's a great testing tool. Genymotion also has a much quicker start time and a much more responsive execution time than the standard Android Emulators. There are free and paid versions depending on what features you need, irrespective of whether you are using Xamarin.Android or native Java Android development. You can find more information about Genymotion on their home page at <http://www.genymotion.com>.

Extending NationalParks.Droid

As we have a good understanding of our starting point, we can now turn our attention to enhance what we have to support the features we need. We have the following enhancements to complete:

1. Add a `ListView` instance to `MainActivity` to list national parks and an `add` action in the `ActionBar` class to add a new national park.
2. Add a detail view that can be used to view and update national parks with actions to save and delete national parks as well as to view photos on www.Bing.com and get directions from a map provider.
3. Add logic to load and save national parks to a JSON-formatted text file.

Storing and loading national parks

Similar to the `NationalParks.iOS` project, we will store our parks data in a JSON-formatted text file. In this project, we will create a singleton class to help manage loading and saving parks. We are going with a singleton class for a couple of reasons:

- In the next chapter, we are going to start looking at sharing and reusing code; this gets us started on a solution we will want to reuse
- It's a little more difficult to pass an object between `Activities` in Android than it is between `ViewControllers` in iOS, and the singleton class will provide a convenient way to share park data

Adding Json.NET

If you worked through *Chapter 4, Developing Your First iOS App with Xamarin.iOS*, Json.NET will already be installed on your machine and you simply need to add this to your project.

To add Json.NET to the `NationalParks.Droid` project, perform the following steps:

1. Select the `Components` folder in the `NationalParks.Droid` project in the **Solution** pad, right-click and choose **Edit Components**.
2. If you see Json.NET listed in the **Installed on this machine** section, click on **Add to Project** and you are done; otherwise continue with the next step.
3. In the upper-right hand corner, click on **Get More Components** and enter `Json.NET` in the search field.
4. Select `Json.NET` from the list and choose **Add to App**.

Borrowing the entity class and JSON file

We need an entity class that represents our subject area: the national parks. This sounds familiar if you worked through *Chapter 4, Developing Your First iOS App with Xamarin.iOS*, where we created one. As one already exists, there is no need to create one from scratch, so let's just copy it from `NationalParks.iOS`. In *Chapter 6, The Sharing Game*, we will look at actually sharing the code between projects.

To copy the `NationalPark.cs` file, perform the following steps:

1. Select the `NationalPark.cs` file in the `NationalParks.iOS` project, right-click on it, and choose **Copy**.
2. Select the `NationalPark.Droid` project, right-click on it and choose **Paste**.

Creating the NationalParksData singleton

As we mentioned previously, we will create a singleton class to simplify sharing and accessing national parks. A singleton is a design pattern that restricts the number of instances of class that can exist within an app to a single instance. Singletons can be helpful in maintaining global state and sharing a single object across multiple views. For our purposes, the singleton pattern simplifies managing a single collection of national parks and housing the logic required to load and save parks to a JSON-formatted file.

To create `NationalParksData`, perform the following steps:

1. Select `NationalParks.Droid`, right-click on it, and navigate to **Add | New File**.
2. Select **General | Empty Class**, enter `NationalParksData` for the **Name** field, and click on **New**.
3. Add a `static instance` property to access a single instance for `NationalParksData` and initialize the single instance in the getter for the property, as follows:

```
private static NationalParksData _instance;
public static NationalParksData Instance
{
    get { return _instance ??
        (_instance = new NationalParksData()); }
}
```

4. Add a `Parks` collection property to load the parks into:

```
public List<NationalPark> Parks { get; protected set; }
```



Note the use of `protected set`, which protects the `Parks` property from being modified outside of the singleton class.

5. There are several places we need to determine the filename to load and save parks to a JSON-formatted file. Create a method that returns a fully-qualified filename, as follows:

```
protected string GetFilename()
{
    return Path.Combine (
        Environment.GetFolderPath(
            Environment.SpecialFolder.MyDocuments),
        "NationalParks.json");
}
```

6. Add a private constructor that loads the `Parks` collection if a file exists. Providing a private constructor is a part of implementing the singleton pattern as it helps ensure only a single instance exists. The private constructor can be added using the following code snippet:

```
private NationalParksData ()
{
    if (File.Exists (GetFilename())) {
        string var serializedParks = File.ReadAllText
            (GetFilename());
        Parks =
            JsonConvert.DeserializeObject<List<NationalPark>>
                (serializedParks);
    }
    else
        Parks = new List<NationalPark> ();
}
```

7. Add a `Save()` method. This method accepts a park, adds it to the `Parks` collection if it is a new park, and then saves the collection to the file, as follows:

```
public void Save(NationalPark park)
{
    if (Parks != null) {
        if (!Parks.Contains (park))
            _parks.Add (park);
        string var serializedParks =
            JsonConvert.SerializeObject (Parks);
        File.WriteAllText (GetFilename (), serializedParks);
    }
}
```

8. Add a `Delete()` method. This method removes the park from the `Parks` collection and saves the updated collection to the file, as follows:

```
public void Delete(NationalPark park)
{
    if (Parks != null) {
        Parks.Remove (park);
        string serializedParks =
            JsonConvert.SerializeObject (Parks);
        File.WriteAllText (GetFilename (), serializedParks);
    }
}
```

Enhancing MainActivity

With the `NationalParksData` singleton in place, we can move on to some of the UI work. The `Xamarin.Android` project template did not give us much of a head start as we received in the last chapter. We need to add a list view to `MainActivity`, create a `DetailActivity` to view a park, and create an `EditActivity` to update and delete parks. `MainActivity` is a good starting point.

Adding a ListView instance

The default view (`Main.xml`) generated when we created the project contains a `Button` instance within `LinearLayout`. We need to remove this button and add a `ListView` instance to display our parks.

Touring the Xamarin.Android Designer

Xamarin Studio provides a graphical design tool to create and edit layout files. As this is our first time using this tool, we will devote a few minutes to become familiar with it. Perform the following steps:

1. Open `Main.xml`; note the two tabs at the bottom of the view, **Content** and **Source**. With the **Content** tab selected, a visual representation of the layout is displayed. With the **Source** tab selected, the raw XML is displayed in an XML editor.
2. Now, switch to the **Content** tab. Note that on the right-hand side of Xamarin Studio, there are two pads, **Document Outline** and **Properties**. When a layout is opened in the **Content** mode, the **Document Outline** pad displays a hierarchical view of the contents of the layout file. The **Document Outline** pad shows the **Button** control within `LinearLayout`.

3. The **Properties** pad displays properties for the currently selected widget. Select the **Button** instance and switch to the **Properties** pad. Note the tabs at the top of the **Properties** pad: **Widget**, **Style**, **Layout**, **Scroll**, and **Behavior**. The tabs group together the various types of properties available for a particular widget.

Editing the Main.xml file

To add a `ListView` instance in `Main.xml`, perform the following steps:

1. With `Main.axml` open in the **Content** mode, select the **Button** instance, right-click on it, and choose **Delete** (or press the *Delete* key).
2. In the search field at the top of the **Toolbox** tab, enter `List`. Select the `ListView` widget displayed and drag-and-drop it to `Main.axml`.
3. In the **Document Outline** pad, select the `ListView` widget.
4. In the **Properties** pad under the **Widget** tab, enter `@+id/parksListView` for the **ID** value.
5. In the **Document Outline** pad, select the `LinearLayout` widget.
6. In the **Properties** pad under the **Layout** tab, enter `8dp` for **Padding**.

Creating an adapter

We need a `ListAdapter` instance to populate our `ListView` with national parks. We will create an adapter that extends from `BaseAdapter`.

To create `NationalParksAdapter.cs`, perform the following steps:

1. Select the `NationalParks.Droid` project, right-click on it, and choose **New File**. In the **New File** dialog box, navigate to **Android | Android Class**.
2. Enter `NationalParks.cs` for the **Name** field and click on **New**.
3. Change `NationalParksAdapter` to be a public class and to extend `BaseAdapter<>` using `NationalPark` as the type specification, as follows:

```
public class NationalParksAdapter :  
    BaseAdapter<NationalPark>  
{  
}  
}
```
4. Place the cursor on `BaseAdapater<>`, right-click on it, and navigate to **Refactor | Implement** abstract members, and then press *Enter*. Xamarin Studio will create a default method stub for each abstract method with code that throws the exception `NotImplementedException`.

5. At this point, you can implement a constructor that accepts an activity and saves the reference for use within `GetView()`, as shown in the following code snippet:

```
private Activity _context;
public NationalParksAdapter(Activity context)
{
    _context = context;
}
```

6. Implement the `GetItemId()` method and return the position that was passed as the ID. The `GetItemId()` method is intended to provide an ID for a row of data displayed in `AdapterView`. Unfortunately, the method must return a `long` instance, and our ID is a GUID. The best we can do is return the position that is passed to us, as follows:

```
public override long GetItemId(int position)
{
    return position;
}
```

7. Implement the `Count` property to return the number of items in the `Parks` collection, as follows:

```
public override int Count
{
    get { return NationalParksData.Instance.Parks.Count; }
}
```

8. Implement the indexed property and return the `NationalPark` instance located at the position passed in within the `Parks` collection, as follows:

```
public override NationalPark this[int position]
{
    get { return NationalParksData.Instance.Parks[position]; }
}
```

9. Implement the `GetView()` method and return a populated `View` instance for a park using the default Android layout `simpleListItem1`, as follows:

```
public override View GetView(int position,
    View convertView, ViewGroup parent)
{
    View view = convertView;
    if (view == null) {
        view =
            _context.LayoutInflater.Inflate(
                Android.Resource.Layout.SimpleListItem1,
```

```
        null) ;  
    }  
  
    view.FindViewById<TextView>  
        (Android.Resource.Id.Text1).Text =  
            NationalParksData.Instance.Parks [position].Name;  
  
    return view;  
}
```

10. To conclude these steps, hook up the adapter to `ListView` on `MainActivity`. This is normally done in the `OnCreate()` method, as follows:

```
NationalParksAdapter _adapter;  
.  
.  
protected override void OnCreate (Bundle bundle)  
{  
    .  
    .  
    .  
    _adapter = new NationalParksAdapter (this);  
    FindViewById<ListView>  
        (Resource.Id.parksListView).Adapter = _adapter;  
    .  
    .  
}
```

Adding the New action to the ActionBar

We now need to add an `Add` action to the `ActionBar`, which can be used to create a new national park.

To create the `Add` action, perform the following steps:

1. You firstly need to select the `Resources` folder in the `NationalParks.Droid` project, right-click on it, and navigate to **Add | New Folder**.
2. At this point, name the folder `menu`.
3. Select the newly created `menu` folder, right-click on it, and navigate to **Add | New File**, then select **XML | Empty XML File**, enter `MainMenu.xml` in the **Name** field, and click on **New**.
4. Fill in the newly created XML file with a menu definition for the `Add` action. The following sample demonstrates what is needed:

```
<menu  
    xmlns:android="http://schemas.android.com/apk/res/android">  
    <item android:id="@+id/actionNew"  
        android:icon="@drawable/ic_new"
```

```
        android:title="New"
        android:showAsAction="ifRoom" />
</menu>
```

5. Copy all of the image files (*.png) from the Assets folder to the Resources/drawable folder in the NationalParks.Droid project.
6. Select the Resources/drawable folder, right-click and choose **Add Files**, select all of the image files, including ic_new.png, and click on **Open**.

Now that we have the menu definition and graphics in place, we need to add some code to put the menus in place. Android provides several virtual methods to create and process clicks for ActionBar items.

Overriding the OnCreateOptionsMenu() method

The `OnCreateOptionsMenu()` method is called when an activity is started and provides a place to create ActionBar items. The following code demonstrates how to use the definition in `MainMenu.xml` to create the Add action:

```
public override bool OnCreateOptionsMenu(IMenu menu)
{
    MenuInflater.Inflate(Resource.Menu.MainMenu, menu);
    return base.OnCreateOptionsMenu(menu);
}
```

Overriding the OnOptionsItemSelected() method

The `OnOptionsItemSelected()` method is called when an action in the ActionBar is clicked on, and it provides a place to handle the request. In our case, we want to navigate to the detail view, which has not been created yet. For now, simply implement the `OnOptionsItemSelected()` method with a placeholder for the navigation logic. The following code demonstrates what is needed:

```
public override bool OnOptionsItemSelected (
    IMenuItem item)
{
    switch (item.ItemId)
    {
        case Resource.Id.actionNew:
            // Navigate to Detail View
            return true;

        default :
            return base.OnOptionsItemSelected(item);
    }
}
```

Running the app

We have completed our enhancements to `MainActivity`. Run the app and review the changes. When you initially start the app, you will notice that `ListView` is empty. You can place the `NationalParks.json` file in the emulator virtual device using the **Android Device Monitor (ADM)**. Xamarin Studio is not configured with a menu item for ADM, but you can add one using **Preferences | External Tools**.

Upload `NationalParks.json` to the emulator using the ADM application. Restart `NationalParks.Droid`; you should now see parks listed.

Creating the DetailActivity view

Now, let's add a view that displays the details for a national park. For this, we need to create a simple view with `ScrollView` as the parent `ViewGroup` and `EditText` widgets for each of the properties on the `NationalPark` entity class.

To create the `DetailActivity` view, perform the following steps:

1. Select the `NationalParks.Droid` project in the **Solution** pad, right-click and navigate to **Add | New File**.
2. After this, navigate to **Android | Android Activity**, enter `DetailActivity` for the value of the **Name** field, and click on **New**.
3. Then, select the `Resources/layout` folder in `NationalParks.Droid`, right-click on it, and navigate to **Add | New File**.
4. Navigate to **Android | Android Layout**, enter `Detail` for the **Name** field, and click on **New**.
5. In the **Outline** pad, select `LinearLayout`, right-click on it, and choose **Delete**.
6. From the **Toolbox** pad, select the `ScrollView` widget and drag it onto the `Detail.axml` layout.
7. From the **Toolbox** pad, select `LinearLayout` and drag it onto the `Detail.axml` layout.
8. In the **Properties** pad under the **Layout** tab, set **Padding** to `8dp` for `LinearLayout`.

9. Add `TextView` widgets for each property on the `NationalPark` entity class except the `ID` property. Also, add `TextView` widgets that serve as labels. For each of the `TextView` widgets that will be used to display properties, fill in the `ID` property with a name that corresponds to the property names on the entity, for example, `nameTextView`. Arrange the widgets based on your preferences; you can use the screen mockups in the *The sample national parks app* section of *Chapter 4, Developing Your First iOS App with Xamarin.iOS*, or the sample solution as a guide.
10. Review `Detail.axml` in the **Content** mode and adjust it as needed.
11. In `DetailActivity.OnCreate()`, add call to `SetContentView()`, and pass the layout ID for `Detail.axml`, as follows:

```
SetContentView (Resource.Layout.Detail);
```

Adding ActionBar items

We now need to add three items to the action bar: an action to edit a park, view photos on www.bing.com for a park, and get directions to a park. Follow the same steps used previously to create a new menu definition file named `DetailMenu.xml`. The following XML shows the code that needs to be used:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/actionEdit"
        android:icon="@drawable/ic_edit"
        android:title="Edit"
        android:showAsAction="ifRoom" />
    <item android:id="@+id/actionPhotos"
        android:title="Photos"
        android:showAsAction="never" />
    <item android:id="@+id/actionDirections"
        android:title="Directions"
        android:showAsAction="never" />
</menu>
```

After adding the menu definition, implement the `OnCreateOptionsMenu()` and `OnOptionsItemSelected()` methods like we did for `MainActivity`. Just add empty stubs to handle the actual actions and we will fill in the logic in the coming sections.

Populating DetailActivity

Add logic to populate `DetailActivity` in `OnCreate()`, using the following steps:

1. The first step is to determine if a park `Id` was passed in as an intent extra. If one was passed in, locate it in the `Parks` list on `NationalParksData`. If not, create a new instance using the following snippet:

```
if (Intent.HasExtra ("parkid")) {  
    string parkId = Intent.GetStringExtra ("parkid");  
    _park = NationalParksData.Instance.  
        Parks.FirstOrDefault (x => x.Id == parkId);  
}  
else  
    _park = new NationalPark ();
```

2. Now populate the `EditText` fields based on data from the park. The sample solution has a `ParkToUI()` method for this logic, as follows:

```
protected void ParkToUI ()  
{  
    _nameEditText.Text = _park.Name;  
    . . .  
    _latEditText.Text = _park.Latitude.ToString();  
    . . .  
}
```

Handling the Show Photos action

We would like to direct the user to `www.bing.com` to view photos for a park. This can be accomplished with a simple `ActionView` intent and a properly formatted search URI for `www.bing.com`.

To handle the Show Photos action, create some logic on the `OnOptionsItemSelected()` method to create an `ActionView` intent and pass in a formatted URI to search `www.bing.com` for photos. The following code demonstrates the required action:

```
public override bool OnOptionsItemSelected (  
    IMenuItem item)  
{  
    switch (item.ItemId) {  
        . . .  
        case Resource.Id.actionPhotos:  
            Intent urlIntent =  
                new Intent (Intent.ActionView);
```

```
urlIntent.SetData (
    Android.Net.Uri.Parse (
        String.Format (
            "http://www.bing.com/images/search?q={0}",
            _park.Name)));
StartActivity (urlIntent);
return true;
.
.
.
}
```

Handling the Show Directions action

We would like to direct the user to an external map app to get directions to a park. Again, this can be accomplished with a simple `ActionView` intent along with a properly formatted URI requesting map information.

To handle the Show Directions action, create a logic on the `OnOptionsItemSelected()` method to create an `ActionView` intent and pass in a formatted URI to display the map information. The following code demonstrates the required action:

```
case Resource.Id.actionDirections:

if ((_park.Latitude.HasValue) &&
    (_park.Longitude.HasValue)) {
    Intent mapIntent = new Intent
        (Intent.ActionView,
        Android.Net.Uri.Parse (
            String.Format ("geo:0,0?q={0},{1}&z=16 ({2})",
            _park.Latitude,
            _park.Longitude,
            _park.Name)));
    StartActivity (mapIntent);
}

return true;
```

Adding navigation

Now that we have `DetailActivity` in place, we need to go back and add some navigation logic in `MainActivity` so that when a park is selected in the list, `DetailActivity` will be displayed.

A user clicking on an item in `ListView` can be handled by providing an event handler for `ListView.OnItemClicked`.

To add navigation from `MainActivity`, perform the following steps:

1. Open `MainActivity.cs`.
2. Create an event handler to handle an `OnItemClicked` event. The following event handler represents what is needed:

```
public void ParkClicked(object sender,
    ListView.ItemClickEventArgs e)
{
    Intent intent = new Intent (this,
        typeof(DetailActivity));
    intent.PutExtra("parkid", adapter[e.Position].Id);
    StartActivity (intent);
}
```

3. Hook up the event handler in the `OnCreate()` method, as follows:

```
FindViewById<ListView>
(Resource.Id.parksListView).ItemClick += ParkClicked;
```

Running the app

We have now completed `DetailActivity`. Run the app and select a park to display the new activity. Choose the Show Photos and Show Directions actions. If you are running the application in an emulator, you will not be able to view directions as the emulator will not have access to Google Play Services.

Creating `EditActivity`

We are now ready to add our last activity, `EditActivity`. This exercise will be similar to the one we just finished except that we will use `EditText` widgets so that users will be able to modify data. In addition, `EditActivity` can be used to display an existing park or a new one.

To create `EditActivity`, perform the following steps:

1. Follow the same steps used in the previous section to create a new activity and layout file named `EditActivity` and `Edit.axml` respectively.
2. Also, add `ScrollView`, `LinearLayout`, and `Padding` in the same manner as was done for `Detail.axml`.

3. Add `TextView` widgets and `EditText` widgets for each property on the `NationalPark` entity class except the `Id` property. The `TextView` widgets should be used as labels and the `EditText` widgets to edit properties. For each of the `EditView` widgets that will be used to display properties, fill in the `Id` property with a name that corresponds to the property names on the entity, for example, `nameTextView`. Arrange the widgets based on your preferences; you can use the screen mockups in the *The sample national parks app* section of *Chapter 4, Developing Your First iOS App with Xamarin.iOS*, or sample solution as a guide.
4. Review `Edit.axml` in the **Content** mode and adjust as needed.
5. In `EditActivity.OnCreate()`, add a call to `SetContentView()` and pass in the layout `Id` for `Edit.axml`, as follows:

```
SetContentView (Resource.Layout.Edit);
```

Adding ActionBar items

We now need to add three items to the action bar: an action to edit a park, view photos on www.bing.com for a park, and get directions to a park. Follow the same steps used previously in the section *Adding the New action to the ActionBar* to create a new menu definition file named `DetailMenu.xml`. The following XML shows the required code:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/actionSave"
        android:icon="@drawable/ic_save"
        android:title="Save"
        android:showAsAction="always" />
    <item android:id="@+id/actionDelete"
        android:icon="@drawable/ic_delete"
        android:title="Delete"
        android:showAsAction="always" />
</menu>
```

After adding the menu definition, implement the `OnCreateOptionsMenu()` and `OnOptionsItemSelected()` methods like we did for `MainActivity`. Add empty stubs to handle each action and we will fill in the logic in the coming sections, as follows:

```
public override bool OnOptionsItemSelected (IMenuItem item)
{
    switch (item.ItemId)
    {
```

```
        case Resource.Id.actionSave:  
            // will add save logic here...  
            return true;  
  
        case Resource.Id.actionDelete:  
            // will add delete logic here...  
            return true;  
  
        default :  
            return base.OnOptionsItemSelected(item);  
    }  
}
```

Creating reference variables for widgets

As we will be putting data in the `EditText` widgets and then pulling it back out again, it makes sense to create reference variables for the widgets and set the references in the `OnCreate()` method.

To create reference variables for widgets, perform the following steps:

1. Create a set of references to `EditText` objects in the `EditActivity` class, as follows:

```
EditText _nameEditText;  
EditText _descrEditText;  
. . .
```

2. In the `OnCreate()` method of `EditActivity`, set the references to the appropriate widgets using `FindViewById()`, as follows:

```
_nameEditText= FindViewById<EditText>  
    (Resource.Id.nameEditText);  
_descrEditText = FindViewById<EditText>  
    (Resource.Id.descrEditText);  
. . .
```

Populating EditActivity

To populate `EditActivity`, perform the following steps:

1. Create a method named `ParkToUI()` to move data from the `_park` object to the `EditText` widgets, as follows:

```
protected void ParkToUI()  
{  
    _nameEditText.Text = _park.Name;
```

```
    _descrEditText.Text = _park.Description;
    . . .
}
```

2. Override `OnResume()` and add a call to the `ToUI()` method to populate the `EditText` widgets, as follows:

```
protected override void OnResume ()
{
    base.OnResume ();
    ParkToUI ();
}
```

Handling the Save action

When the `Save` action is clicked on, the `OnOptionsItemSelected()` method is called. Create a `Save()` method on `DetailActivity` and call it from `OnOptionsItemSelected()`. The solution project has a `UIToPark()` method to take content from the `EditText` widgets and populate the `Park` entity before saving it.

To handle the `Save` action, perform the following steps:

1. Create a method named `ToPark()` to move data from the `EditText` widgets to the `_park` object. This method will be used when handling the `Save` action, as follows:

```
protected void UIToPark()
{
    _park.Name = _nameEditText.Text;
    _park.Description = _descrEditText.Text;
    . . .
    if (!String.IsNullOrEmpty (_latEditText.Text))
        _park.Latitude = Double.Parse (_latEditText.Text);
    else
        _park.Latitude = null;
    . . .
}
```

2. Create a method to handle saving the park that calls `UIToPark()` to populate the `_park` object with changes, then it calls the `Save()` method on `NationalParksData` to save the changes to file, sets the result code, and finishes the activity. The required code is as follows:

```
protected void SavePark()
{
    UIToPark ();
}
```

```
NationalParksData.Instance.Save (_park);  
  
Intent returnIntent = new Intent ();  
returnIntent.PutExtra ("parkdeleted", false);  
SetResult (Result.Ok, returnIntent);  
  
Finish ();  
}
```



Note that a Boolean Extra named parkdeleted is set to false. This is used to communicate to the calling activity that the park was not deleted.



3. Update `OnOptionsItemSelected()` to call `SavePark()`, as follows:

```
case Resource.Id.actionSave:  
    SavePark ();  
    return true;
```

Handling the Delete action

Handling the Delete action is similar to the Save action, but somewhat simpler as we do not have to save changes from the UI widgets.

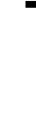
To handle the Delete action, perform the following steps:

1. Create a method to handle the deleting of the park by calling the `Delete()` method on `NationalParksData`, setting the result code, and finishing the activity, as follows:

```
protected void DeletePark()  
{  
    NationalParksData.Instance.Delete (_park);  
  
    Intent returnIntent = new Intent ();  
    returnIntent.PutExtra ("parkdeleted", true);  
    SetResult (Result.Ok, returnIntent);  
  
    Finish ();  
}
```



Note that the Boolean Extra named parkdeleted is set to true to tell the calling activity that the park was deleted. This is important to the `DetailActivity` because when a park was previously shown as deleted, it should be finished and returned to `MainActivity`.



2. Update `OnOptionsItemSelected()` to call `DeletePark()`, as follows:

```
case Resource.Id.actionDelete:
    DeletePark ();
    return true;
```

Adding navigation

Now that `EditActivity` is in place, we need to add navigation logic to `MainActivity` when a user chooses the `New` action and to `DetailActivity` when the user chooses the `Edit` action.

Navigating on the New action

As you can recall, in `OnMenuItemSelected()` in `MainActivity`, we added a comment to the place where we need to navigate to `EditActivity`. We can now replace this comment with the following use of `StartActivity()`:

```
public override bool OnOptionsItemSelected (
    IMenuItem item)
{
    switch (item.ItemId)
    {
        case Resource.Id.actionNew:
            StartActivity (typeof(DetailActivity));
            return true;
        default :
            return base.OnOptionsItemSelected(item);
    }
}
```

Navigating on the Edit action

In the same way, we need to add navigation code to `OnMenuItemSelected()` in `DetailActivity`. However, there are a few differences. We need to pass in the `Id` property for the park we want to edit and we want to receive back a result that indicates whether or not the user deleted this park. The required code is as follows:

```
case Resource.Id.actionEdit:
    Intent editIntent = new Intent(this,
        typeof(EditActivity));
    editIntent.PutExtra("parkid", _park.Id);
    StartActivityForResult(editIntent, 1);
    return true;
```

`DetailActivity` also needs to detect when a park is deleted so that it can finish and return to `MainActivity` to view the list. To accomplish this, override `OnActivityResult()` and check the Boolean `Extra` named `parkdeleted` to determine if the park was deleted, as follows:

```
protected override void OnActivityResult (
    int requestCode, Result resultCode, Intent data)
{
    if ((requestCode == 1) && (resultCode == Result.Ok))
    {
        if (data.GetBooleanExtra ("parkdeleted", false))
            Finish ();
    }
    else
        base.OnActivityResult (
            requestCode, resultCode, data);
}
```

Refreshing ListView in MainActivity

The last thing we need to implement is the logic that will refresh `ListView` in `MainActivity` with any changes that might have been made on `EditActivity`. To accomplish this, call `NotifyDataSetChanged()` on the adapter object within an override to the `OnResume()` method on `MainActivity`, as follows:

```
protected override void OnResume ()
{
    base.OnResume ();
    adapter.NotifyDataSetChanged ();
}
```

Running the app

We have now completed the `NationalParks.Droid` app. You should now be able to run your app and exercise each of the features.

Working with Xamarin.Android projects in Visual Studio

If you have installed Xamarin.Android on a Windows machine with Visual Studio 2010 or Visual Studio 2013 (which is the current version), the Xamarin.Android Visual Studio add-on will already be installed. Working with projects in Visual Studio is similar to working with Xamarin Studio with the exception of certain features. To access options for the project, perform the following steps:

1. Select the `NationalParks.Droid` project, right-click and select **Properties**. A multi-tabbed window will be open that allows various project-related options to be specified.
2. To access Xamarin.Android-related options for Visual Studio, navigate to **Tools | Options | Xamarin | Android Settings**.
3. To access the AVD Manager, navigate to **Tools | Open Android Emulator Manager**.
4. To manage your Xamarin account and activate a license, navigate to **Tools | Xamarin Account**.

If you are working on a Windows machine with Visual Studio installed and you have not taken time to try out the add-on, open `NationalParks.Droid` in Visual Studio and run the app.

Reviewing the generated elements

Prior to wrapping this chapter up, let's look at some of the things that go on behind the scenes.

Peer objects

In *Chapter 3, Demystifying Xamarin.Android*, we discussed the role of peer objects in a Xamarin.Android app. Let's now take a look at one of the generated Java peer objects from our project. The source for these classes can be found in `NationalParks.Droid/obj/Debug/android/src`. Open `nationalparks.droid.MainActivity.java`. Now, note the following pointers:

- `MainActivity` extends `android.app.Activity`.

- Each method we created an override for has a corresponding method created that calls our override. For example, we created an override for `OnCreate()`. The generated class has a method named `onCreate()` that calls a private native method `n_onCreate()`, which in turn points to our override through a JNI reference.
- The static class initializer for `MainActivity` registers all the native methods for use with JNI using the `mono.android.Runtime.register()` method.
- The class constructor activates an instance of our managed C# class using the `mono.android.TypeManager.Activate()` method.

The `AndroidManifest.xml` file

Xamarin.Android generates an `AndroidManifest.xml` file at build time using two sources as input: the first one being the content in the `AndroidManifest.xml` file in `NationalParks.Droid/Properties` and the second one being the attributes specified on classes, primarily activities in your project. You can find the generated `AndroidManifest.xml` in `NationalParks.Droid/obj/Debug/android`. Open the file with a text editor and note the following pointers:

- There are two `<activity/>` elements in the file and `MainActivity` is specified to be the launch activity. These entries are generated from the attributes specified on each of the activity classes.
- A single permission of `INTERNET` is specified. This came from the `AndroidManifest.xml` file in the `NationalParks.Droid/Properties` folder.

The APK file

Another interesting thing to look at is the APK produced for a Xamarin.Android app. We will be covering in detail how to create APKs in *Chapter 10, Preparing Xamarin.Android Apps for Distribution*. This a fairly simple process; if you can't wait, use the following steps:

1. In the upper-left hand corner of the toolbar, set the built type to **Release**.
2. From the **Project** menu, select **Publish Android Project**.
3. In the **Publish Android Application** dialog box, choose **Create new keystore**, fill out all of the required information, and click on **Create**.
4. Xamarin.Android will publish the APK in the location you selected. As APKs are ZIP files, simply unzip the APK to view the contents.

The following screenshot shows the contents of the resulting APK:

Name	Size
AndroidManifest.xml	2 KB
assemblies	--
Mono.Android.dll	662 KB
Mono.Security.dll	155 KB
mscorlib.dll	1.4 MB
NationalParks.Droid.dll	11 KB
Newtonsoft.Json.dll	343 KB
System.Core.dll	56 KB
System.dll	269 KB
System.Runtime.Serialization.dll	8 KB
classes.dex	232 KB
environment	Zero bytes
lib	--
armeabi-v7a	--
libmonodroid.so	2.9 MB
META-INF	--
NOTICE	157 bytes
res	--
drawable	--
layout	--
detail.xml	3 KB
main.xml	676 bytes
menu	--
resources.arsc	2 KB

The following table provides a description of the contents of the APK:

Content	Description
assemblies/System.*	These assemblies contain core .NET namespaces such as System.IO and System.Collection
assemblies/Mono.Android.dll	This assembly contains the Xamarin.Android binding classes
assemblies/NationalParks.Droid.dll	This assembly contains the classes we created: MainActivity, DetailActivity, and NationalParksAdapter
assemblies/Newtonsoft.Json.dll	This assembly contains the Json.NET classes
classes.dex	This file contains all the generated Java peer objects in a Dalvik-compiled format
lib/armeabi-v7a/libmonodroid.so	This is the Mono CLR for Android
res/*	This folder contains all the resources; drawables, layouts, menus, and so on

Summary

In this chapter, we created a sample Xamarin.Android app and demonstrated the concepts that need to be understood to work with the Xamarin.Android platform. While we did not demonstrate all of the features that can be used in an Android app, you should now feel comfortable with how to access these features.

In the next chapter, we will turn our attention to the important topics of sharing code across apps, one of the key advantages of using Xamarin.

6

The Sharing Game

In this chapter, we will discuss one of the most interesting and important aspects of developing with Xamarin: cross-platform code sharing. We will cover the following topics:

- The file linking technique
- Portable Class Libraries
- The pros and cons of each approach

Sharing and reuse

One of the advantages of using Xamarin and C# is the ability to share code across your mobile apps as well as other .NET solutions. The reuse of code can provide significant productivity and reliability advantages as well as reduce many of the long-term maintenance headaches that come with long-lived apps. That's great, but anyone who has been involved in software development for a long period of time understands that reuse is not free and not simple to achieve.

There are practical aspects of reuse; the question is, "Physically, how do I package my code for reuse?" For this, we can use one of the following three methods:

- Share source code that can be compiled into multiple projects
- Share **Dynamic-Link Library (DLL)** that can be referenced by multiple projects
- Share code as a service that can be accessed remotely by multiple clients

There are also more strategic aspects; again the question arises, "How can I organize my code so that I can reuse more of it?" To solve this problem, we have the following options:

- Create a layered approach so that data access logic and business validation is separated out of the user interface logic
- Utilize interfaces and frameworks that abstract platform-specific services away from the reusable layers

In this chapter, we will touch on both these aspects of reuse, but primarily focus on the practical side of reuse. Specifically, we will present two different approaches to bundle up the code for reuse.

So, what parts of our code should we try and reuse? In the work we have done on the `NationalParks` apps so far, one obvious set of code stands out for reuse: the persistence code, which is the logic that loads parks from a JSON file and saves them back to the same file. In *Chapter 5, Developing Your First Android App with Xamarin.Android*, we moved towards a reusable solution by creating the `NationalParkData` singleton. In this chapter, we will demonstrate two different methods for sharing the `NationalParkData` singleton across both our projects as well as other .NET projects that might need it.

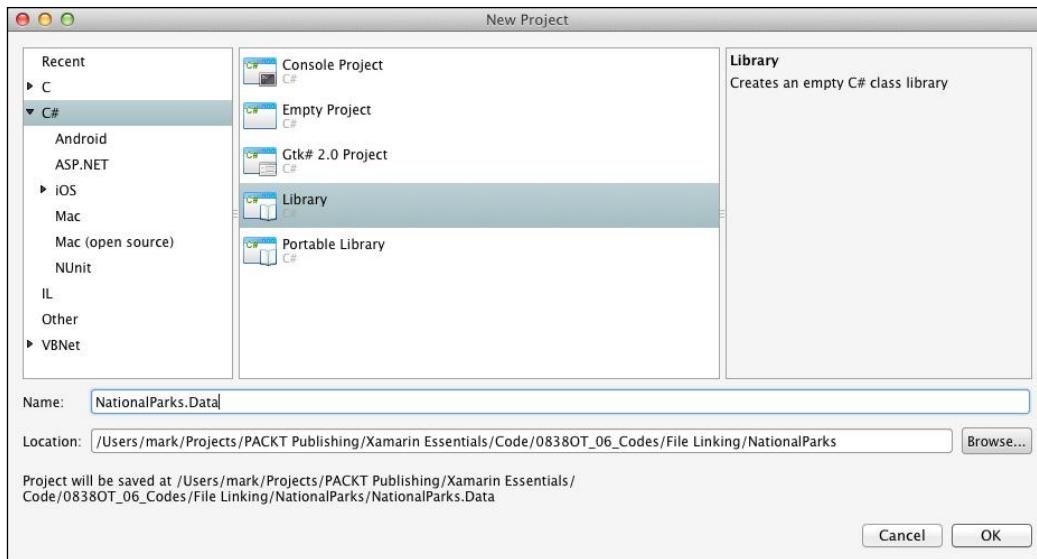
Old school source file linking

File linking refers to a technique where source code files are linked or referenced by a Xamarin project and are compiled when a build is run on the project along with the rest of the source code in the project. When using file linking, a separate DLL is not created for the files you are sharing, rather the code is compiled into the same DLL produced for the project that the file is linked to; in our case, either `NationalParks.ios.dll` or `NationalParks.Droid.dll`.

Creating a shared library project

We will start by creating a new Library project to house the reusable code. To create a Library project, perform the following steps:

1. Add a new library project with the name `NationalParks.Data` to the `NationalParks` solution. You can find the Library project template in the **New Project** dialog box under **C# | Library**, as shown in the following screenshot:



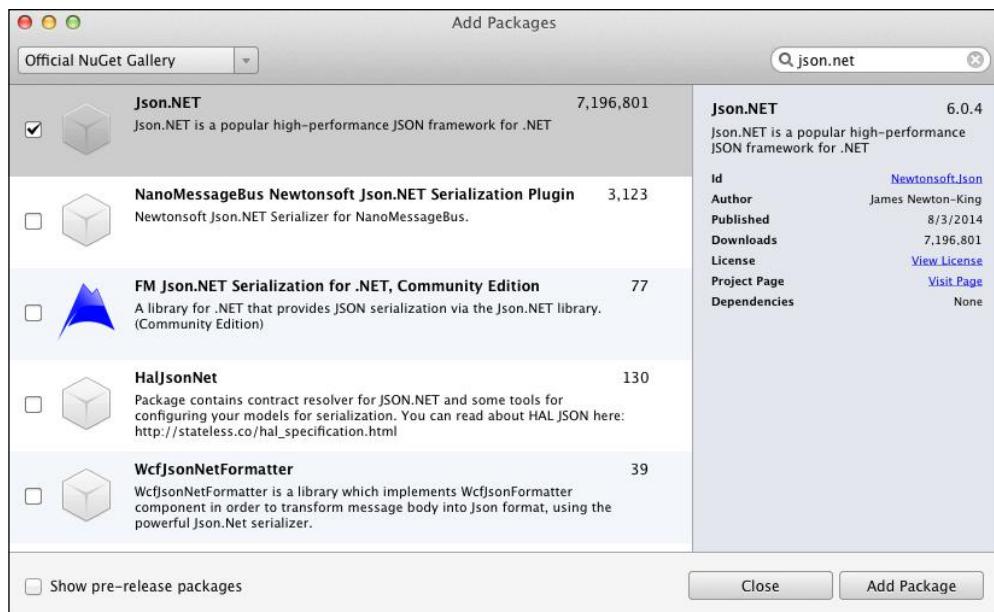
2. Remove `MyClass.cs` from the new project. When removing the file, selecting **Delete** will remove the file from being referenced by the project and delete the underlying file from the filesystem.
3. Set the **Target Framework** option to **Mono/.NET 4.5** in the **Project Options** dialog box under **Build | General**.
4. Move the `NationalPark.cs` and `NationalParkData.cs` files from `NationalParks.Droid` to `NationalPark.Data`.
5. Open `NationalPark.cs` and `NationalParkData.cs` and change the namespace to `NationalParks.Data`.
6. Add a public string `DataDir` property to `NationalParkData` and use it in the `GetFilename()` method, as follows:

```
public string DataDir { get; set; }
.
.
protected string GetFilename()
{
    return Path.Combine (DataDir, "NationalParks.json");
}
```

7. Move the logic to load the parks data from the constructor to a new method named `Load()`, as shown in the following code snippet:

```
public void Load()
{
    if (File.Exists (GetFilename ())) {
        string serializedParks =
            File.ReadAllText (GetFilename ());
        _parks = JsonConvert.DeserializeObject
            <List<NationalPark>> (serializedParks);
    }
    else
        _parks = new List<NationalPark> ();
}
```

8. Compile `NationalParks.Data`. You will receive compile errors due to unresolved references to `Json.NET`. Unfortunately, we cannot simply add a reference to the component version of `Json.NET` that we previously downloaded from the Xamarin component store because this version is built to be used with the `Xamarin.iOS` and `Xamarin.Android` profiles and is not binary compatible with `Mono/.NET 4.5` library projects.
9. Add the `Json.NET` library to the project using NuGet. Select `NationalParks.Data`, right-click on it, and navigate to **Add | Add Packages**. Enter `Json.NET` in the search field, check the `Json.NET` entry in the list, and select **Add Packages**. The following screenshot shows the **Add Packages** dialog box:



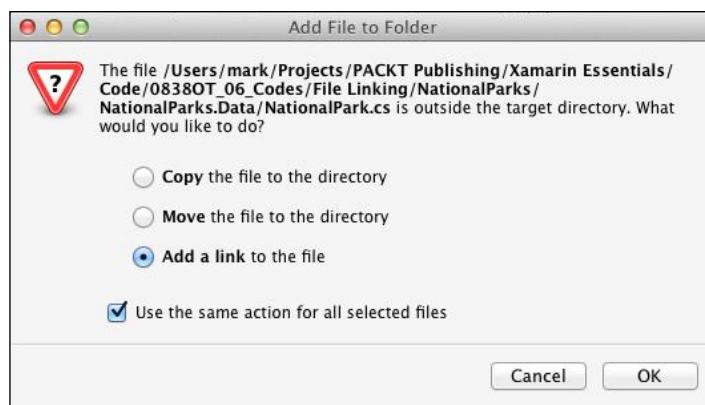
10. Compile NationalParks.Data; you should receive no compile errors this time.

Updating NationalParks.Droid to use shared files

Now that we have the NationalParksData singleton in a separate project, we are now ready to reuse it.

To update NationalParks.Droid in order to use the shared solution, perform the following steps:

1. Select NationalPark.cs and NationalParksData.cs in the **Solution** pad, right-click on it, select **Remove**, and then select **Delete**. This will remove the selected files from the project and physically delete them from the project folder.
2. In NationalParks.Droid, add a folder named NationalParks.Data. This folder will not contain any files, but will simply be used within the project structure to organize links to the shared files.
3. Select the NationalParks.Data folder, right-click on it, and navigate to **Add | Add Files** to add the existing files to the project.
4. In the **Add Files** dialog box, navigate to the NationalParks.Data project folder, select NationalPark.cs and NationalParkData.cs, and click on **Open**.
5. In the **Add File to Folder** dialog box, select **Add a link to file**, check the **Use the same action for all selected files** option, and click on **OK**. Expand the NationalParks.Data folder to see that two file links are added. The following screenshot shows the **Add File to Folder** dialog box:



6. Add a `using` clause to the `NationalParks.Data` namespace and remove any `using` directives for `Newtonsoft.Json` in `MainActivity`, `DetailActivity`, `EditActivity`, and `NationalParksAdapter`.
7. In `MainActivity.OnCreate()`, set the `NationalParksData.DataDir` property and call the `Load()` method prior to creating the `ListView` adapter:

```
NationalParksData.Instance.DataDir =  
    System.Environment.GetFolderPath (  
        System.Environment.SpecialFolder.MyDocuments) ;  
NationalParksData.Instance.Load () ;
```

8. Compile and run the app. You should see no noticeable behavior changes, but we are now using the serialization and storage logic in a shareable way.

Updating NationalParks.iOS to use shared files

Now, let's move on to update `NationalParks.iOS`. We have a little more work to do here because if you can recall, we had the file handling logic spread out in several areas.

To update `NationalParks.iOS` in order to use the shared solution, perform the following steps:

1. Remove `NationalPark.cs` from the project.
2. Add a folder named `NationalParks.Data` in the `NationalParks.Droid` project.
3. Add file links to `NationalPark.cs` and `NationalParksData.cs`.
4. Open `MasterViewController.cs`, add a `using` instance of `NationalParks.Data`, and remove the `using` instance of `Newtonsoft.Json`.
5. In `MasterViewController.ViewDidLoad()`, set the `DataDir` property before creating the data source for `UITableView`:

```
NationalParksData.Instance.DataDir =  
    Environment.CurrentDirectory;  
NationalParksData.Instance.Load () ;
```

6. In the `DataSource` class, remove the `Parks` collection and remove the loading action of the `Parks` collection in the constructor.

7. Update the methods in `DataSource` to reference the `Parks` collection property in `NationalParksData`.
8. Remove the `Parks` property from `DataSource` and update `MasterViewController.PrepareForSegue()` to use the `Parks` property in `NationalParksData`.
9. Open `DetailViewController` and add a `using` instance of `NationalParks`.
`Data`.
10. In `SetNavController()`, remove the `Parks` collection argument, corresponding private variable, and then update the navigation logic in `MasterViewController`.
11. Open `EditViewController` and add a `using` directive for `NationalParks`.
`Data`.
12. In `SetNavController()`, remove the `Parks` collection argument, corresponding private variable, and then update the navigation logic in `MasterViewController` and `DetailViewController` so that no `Parks` collection is passed in.
13. Remove the `SaveParks()` method.
14. In `DoneClicked()`, replace the logic that adds the park to the collection and saves the collection with a call to `NationalParksData.Instance.Save()`, as follows:

```
private void DoneClicked (object sender, EventArgs e)
{
    ToPark ();
    NationalParksData.Instance.Save (_park);
    NavigationController.PopViewControllerAnimated (true);
}
```

15. In `DeleteClicked()`, replace the logic that removes the park from the collection and saves the collection with a call to `NationalParks.Instance.Delete()`, as follows:

```
partial void DeleteClicked (UIButton sender)
{
    NationalParksData.Instance.Delete (_park);
    NavigationController.PopToRootViewControllerAnimated (true);
}
```

16. Compile and run the app. As with `NationalParks.Droid`, you should see no noticeable behavior changes.

Portable Class Libraries

Portable Class Libraries (PCL) are libraries that conform to a Microsoft standard and can be shared in a binary format across many different platforms such as Windows 7 desktop, Windows 8 desktop, Windows 8 phone, Xbox 360, and Mono. The big advantage with a PCL is that you can share a single binary for all these platforms and avoid distributing source code. However, there are some significant challenges.

One issue we face straightaway is the fact that our code uses APIs that are not supported across all the platforms; specifically `File.Exists()`, `File.ReadAllText()`, and `File.WriteAllText()`. It seems surprising, but most of `System.IO` is not common across all of the .NET profiles; so, the file I/O logic can be difficult to deal with within the shared code. In our case, there are only three methods and we can easily abstract this logic away from the shared code by creating an IO interface. Each platform that uses our shared solution will be responsible for providing an implementation of the IO interface.

Creating NationalParks.PortableData

The first step is to create the Portable Class Library to house our shared solution. To create `NationalParks.PortableData`, perform the following steps:

1. Add a new Portable Class Library project to the `NationalParks` solution. The project template can be found under **C# | Portable Library**.
2. Remove `MyClass.cs` from the newly created project.
3. Copy `NationalPark.cs` and `NationalParksData.cs` from the `NationalParks.Data` project to `NationalParks.PortableData`.
4. Add a reference to the `Json.NET` Portable Class Library.
5. Create the `IFileHandler` interface and add three methods that abstract the three IO methods we need. It will be best to make the read and write methods asynchronous returning `Task<>`, because many of the platforms only support asynchronous IO. This will simplify implementing the interface on these platforms. The following code demonstrates the required action:

```
public interface IFileHandler
{
    bool FileExists (string filename);
    Task<string> ReadAllText (string filename);
    Task WriteAllText (string filename, string content);
}
```

6. Add a public `IFileHandler` property to `NationalParksData` and change all the logic to use this property rather than using `System.IO.File`, as follows:

```

public IFileHandler FileHandler { get; set; }

. . .

public async Task Load()
{
    if (FileHandler.FileExists (GetFilename())) {
        string serializedParks =
            await FileHandler.ReadAllText (GetFilename());
        Parks = JsonConvert.DeserializeObject
            <List<NationalPark>> (serializedParks);
    }
    . . .
}

. . .

public Task Save(NationalPark park)
{
    . . .

    return FileHandler.WriteAllText (
        GetFilename (), serializedParks);
}
public Task Delete(NationalPark park)
{
    . . .

    return FileHandler.WriteAllText (
        GetFilename (), serializedParks);
}

```

Implementing `IFileHandler`

We now need to create an implementation of `IFileHandler` that can be used by both our projects. We will share the file handler implementation using the file linking method from the previous sections.

To implement `IFileHandler`, perform the following steps:

1. In the `NationalParks` solution, create a new Library project named `NationalParks.IO` and set the **Target framework** option to **Mono/.NET 4.5**. This will serve as a shared project for our file handler implementation.
2. Remove the `MyClass.cs` file created by default and add a reference to `NationalParks.PortableData`. This will give us access to the `IFileHandler` interface we intend to implement.

3. Create a class named `FileHandler` in `NationalParks.IO`. Add a `using` directive for the `NationalParks.PortableData` namespace and specify that the class implements the `IFileHandler` interface.
4. Use the **Implement** interface menu item under **Refactor** to create stub implementations for each method on the interface.
5. Implement each of the stub methods. The following code demonstrates the required action:

```
#region IFileHandler implementation
public bool FileExists (string filename)
{
    return File.Exists (filename);
}
public async Task<string> ReadAllText (string filename)
{
    using (StreamReader reader =
        File.OpenText(filename)) {
        return await reader.ReadToEndAsync();
    }
}
public async Task WriteAllText (string filename,
    string content)
{
    using (StreamWriter writer =
        File.CreateText (filename)) {
        await writer.WriteLineAsync (content);
    }
}
#endregion
```

Updating NationalParks.Droid to use PCL

Now, it's time to update `NationalParks.Droid` in order to use our new PCL.

To update `NationalParks.Droid` in order to use `NationalParks.PortableData`, perform the following steps:

1. In the `NationalParks.Droid` project, remove the `NationalParks.Data` folder, create a new folder named `NationalParks.IO`, and add a reference to `NationalParks.PortableData`.
2. In the `NationalParks.IO` folder, add **Link** to the `FileHandler` class.
3. In `MainActivity.cs`, add a `using` clause for `NationalParks.IO` and `NationalParks.PortableData`.

4. In `MainActivity.OnCreate()`, initialize the `FileHandler` property with an instance of `FileHandler`, place an `await` instance on the call to `Load()`, and move the assignment of `NationalParksAdapter` before the call to `Load()`, as shown in the following code snippet:

```
_adapter = new NationalParksAdapter (this);
NationalParksData.Instance.FileHandler =
    new FileHandler ();
NationalParksData.Instance.DataDir =
    System.Environment.GetFolderPath (
        System.Environment.SpecialFolder.MyDocuments);
await NationalParksData.Instance.Load ();
```

5. Now that we are loading data asynchronously, the `OnPause()` method will likely be called before the asynchronous return of `OnCreate()`. Thus we need to add a null check for the logic in `OnPause()` that calls `NotifyDataSetChanged()`, as follows:

```
protected override void OnResume ()
{
    base.OnResume ();
    if (_adapter != null)
        _adapter.NotifyDataSetChanged ();
}
```

6. In `NationalParksAdapter.cs`, `DetailActivity.cs`, and `EditActivity.cs`, add a using clause for `NationalParks.PortableData`, and remove the using directive for `NationalParks.Data`.
7. Compile and run the app.

Updating NationalParks.iOS to use PCL

Now, it's time to update `NationalParks.iOS`. For the most part, we go through essentially the same steps.

To update `NationalParks.iOS` in order to use `NationalParks.PortableData`, perform the following steps:

1. In the `NationalParks.Droid` project, remove the `NationalParks.Data` folder, create a new folder named `NationalParks.IO`, and add a reference to `NationalParks.PortableData`.
2. In the `NationalParks.IO` folder, add **Link** to the `FileHandler` class.
3. In `MasterViewController.cs`, add a using clause for `NationalParks.IO` and `NationalParks.PortableData`, and remove the using directive for `NationalParks.Data`.

4. In `MasterViewController.ViewDidLoad()`, initialize the `FileHandler` property with an instance of `FileHandler`, place an `await` instance on the call to `Load()`, and place a call to `TableView.ReloadData()` after the assignment of the data source, as shown in the following code snippet:

```
NationalParksData.Instance.FileHandler =  
    new FileHandler ();  
NationalParksData.Instance.DataDir =  
    Environment.CurrentDirectory;  
await NationalParksData.Instance.Load ();  
TableView.Source = dataSource = new DataSource (this);  
TableView.ReloadData ();
```

5. In `DetailViewController.cs` and `EditViewController.cs`, replace the `using` directive for `NationalParks.Data` with `NationalParks.PortableData`.
6. Compile and run the app.

The pros and cons of the code-sharing techniques

Now that we have some experience with two practical methods for sharing code across Xamarin.iOS and Xamarin.Android apps, let's look at some pros and cons. The following table summarizes some of the pros and cons of each approach:

	Pros	Cons
File linking	<ul style="list-style-type: none">• This allows for a broader use of .NET APIs, assuming that these APIs are supported by all the platforms that will use the shared code. If you are only targeting Xamarin.iOS and Xamarin.Android, this works pretty well.	<ul style="list-style-type: none">• This requires source code to be shared.• These API dependency issues might not be known until shared code has been compiled for each target platform.
Portable Class Library	<ul style="list-style-type: none">• This ensures platform API compatibility.• This allows for distribution of binary code.	<ul style="list-style-type: none">• This limits the namespaces and APIs available for use in your code.

Summary

In this chapter, we reviewed two practical approaches to share code across Xamarin projects as well as other .NET solutions. In the next chapter, we will investigate MvvmCross, a framework that simplifies implementing the Model-View-ViewModel design pattern, increasing the amount of shared code across platforms.

7

Sharing with MvvmCross

In the previous chapter, we covered the basic approaches to reusing code across projects and platforms. In this chapter, we will take the next step and look at how the use of design patterns and frameworks can increase the amount of code that can be reused. We will cover the following topics:

- An introduction to MvvmCross
- The MVVM design pattern
- Core concepts
- Views, ViewModels, and commands
- Data binding
- Navigation (ViewModel to ViewModel)
- The project organization
- The startup process
- Creating `NationalParks.MvvmCross`

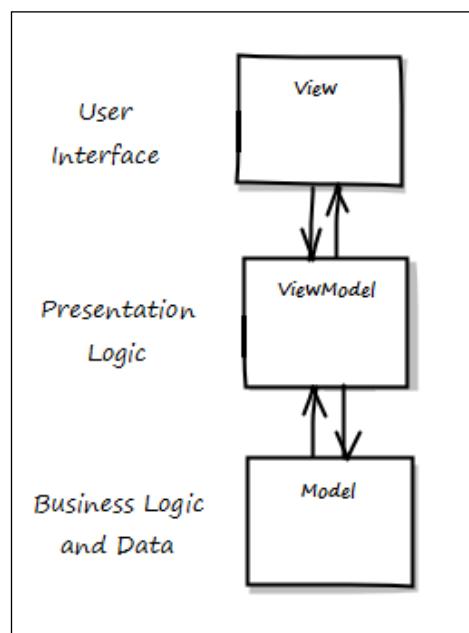
It's more than a little ambitious to try to cover MvvmCross along with a working example in a single chapter. Our approach will be to introduce the core concepts at a high level and then dive in and create the national parks sample app using MvvmCross. This will give you a basic understanding of how to use the framework and the value associated with its use. With that in mind, let's get started.

Introducing MvvmCross

MvvmCross is an open source framework that was created by Stuart Lodge. It is based on the **Model-View-ViewModel (MVVM)** design pattern and is designed to enhance code reuse across numerous platforms, including Xamarin, Android, Xamarin.iOS, Windows Phone, Windows Store, WPF, and Mac OS X. The MvvmCross project is hosted on GitHub and can be accessed at <https://github.com/MvvmCross/MvvmCross>.

The MVVM pattern

MVVM is a variation of the Model-View-Controller pattern. It separates logic traditionally placed in a **View** object into two distinct objects, one called **View** and the other called **ViewModel**. The View is responsible for providing the user interface and the ViewModel is responsible for the presentation logic. The presentation logic includes transforming data from the Model into a form that is suitable for the user interface to work with and mapping user interaction with the View into requests sent back to the Model. The following diagram depicts how the various objects in MVVM communicate:



While MVVM presents a more complex implementation model, there are significant benefits of it, which are as follows:

- ViewModels and their interactions with Models can generally be tested using frameworks (such as NUnit) that are much easier than applications that combine the user interface and presentation layers
- ViewModels can generally be reused across different user interface technologies and platforms

These factors make the MVVM approach both flexible and powerful.

Views

Views in an MvvmCross app are implemented using platform-specific constructs. For iOS apps, Views are generally implemented as ViewControllers and XIB files. MvvmCross provides a set of base classes, such as `MvxViewController`, that iOS ViewControllers inherit from. Storyboards can also be used in conjunction with a custom presenter to create Views; we will briefly discuss this option in the section titled *Implementing the iOS user interface* later in this chapter.

For Android apps, Views are generally implemented as `MvxActivity` or `MvxFragment` along with their associated layout files.

ViewModels

ViewModels are classes that provide data and presentation logic to views in an app. Data is exposed to a View as properties on a ViewModel, and logic that can be invoked from a View is exposed as commands. ViewModels inherit from the `MvxViewModel` base class.

Commands

Commands are used in ViewModels to expose logic that can be invoked from the View in response to user interactions. The command architecture is based on the `ICommand` interface used in a number of Microsoft frameworks such as **Windows Presentation Foundation (WPF)** and Silverlight. MvvmCross provides `IMvxCommand`, which is an extension of `ICommand`, along with an implementation named `MvxCommand`.

The commands are generally defined as properties on a ViewModel. For example:

```
public IMvxCommand ParkSelected { get; protected set; }
```

Each command has an action method defined, which implements the logic to be invoked:

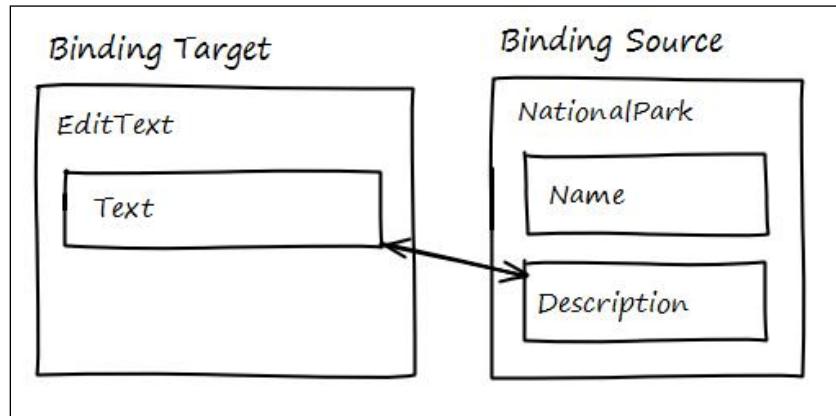
```
protected void ParkSelectedExec(NationalPark park)
{
    . . .// logic goes here
}
```

The commands must be initialized and the corresponding action method should be assigned:

```
ParkSelected =
new MvxCommand<NationalPark> (ParkSelectedExec);
```

Data binding

Data binding facilitates communication between the View and the ViewModel by establishing a two-way link that allows data to be exchanged. The data binding capabilities provided by MvvmCross are based on capabilities found in a number of Microsoft XAML-based UI frameworks such as WPF and Silverlight. The basic idea is that you would like to bind a property in a UI control, such as the **Text** property of an **EditText** control in an Android app to a property of a data object such as the **Description** property of **NationalPark**. The following diagram depicts this scenario:



The binding modes

There are four different binding modes that can be used for data binding:

- **OneWay binding:** This mode tells the data binding framework to transfer values from the ViewModel to the View and transfer any updates to properties on the ViewModel to their bound View property.
- **OneWayToSource binding:** This mode tells the data binding framework to transfer values from the View to the ViewModel and transfer updates to View properties to their bound ViewModel property.
- **TwoWay binding:** This mode tells the data binding framework to transfer values in both directions between the ViewModel and View, and updates on either object will cause the other to be updated. This binding mode is useful when values are being edited.
- **OneTime binding:** This mode tells the data binding framework to transfer values from ViewModel to View when the binding is established; in this mode, updates to ViewModel properties are not monitored by the View.

The `INotifyPropertyChanged` interface

The `INotifyPropertyChanged` interface is an integral part of making data binding work effectively; it acts as a contract between the source object and the target object. As the name implies, it defines a contract that allows the source object to notify the target object when data has changed, thus allowing the target to take any necessary actions such as refreshing its display.

The interface consists of a single event—the `PropertyChanged` event—that the target object can subscribe to and that is triggered by the source if a property changes. The following sample demonstrates how to implement `INotifyPropertyChanged`:

```
public class NationalPark : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
        PropertyChanged;
    // rather than use "... code" it is safer to use
    // the comment form
    string _name;
    public string Name
    {
```

```
    get { return _name; }
    set
    {
        if (value.Equals (_name,
                          StringComparison.OrdinalIgnoreCase))
        {
            // Nothing to do - the value hasn't changed;
            return;
        }
        _name = value;
        OnPropertyChanged();
    }
}
. .
void OnPropertyChanged(
    [CallerMemberName] string propertyName = null)
{
    var handler = PropertyChanged;
    if (handler != null)
    {
        handler(this,
                 new PropertyChangedEventArgs(propertyName));
    }
}
```

Binding specifications

Bindings can be specified in a couple of ways. For Android apps, bindings can be specified in layout files. The following example demonstrates how to bind the `Text` property of a `TextView` instance to the `Description` property in a `NationalPark` instance:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/descrTextView"
    local:MvxBind="Text Park.Description" />
```

For iOS, binding must be accomplished using the binding API. `CreateBinding()` is a method than can be found on `MvxViewController`. The following example demonstrates how to bind the `Description` property to a `UILabel` instance:

```
this.CreateBinding (this.descriptionLabel).
    To ((DetailViewModel vm) => vm.Park.Description).
    Apply ();
```

Navigating between ViewModels

Navigating between various screens within an app is an important capability. Within a MvvmCross app, this is implemented at the ViewModel level so that navigation logic can be reused. MvvmCross supports navigation between ViewModels through use of the `ShowViewModel<T>()` method inherited from `MvxNavigatingObject`, which is the base class for `MvxViewModel`. The following example demonstrates how to navigate to `DetailViewModel`:

```
ShowViewModel<DetailViewModel>();
```

Passing parameters

In many situations, there is a need to pass information to the destination ViewModel. MvvmCross provides a number of ways to accomplish this. The primary method is to create a class that contains simple public properties and passes an instance of the class into `ShowViewModel<T>()`. The following example demonstrates how to define and use a `parameters` class during navigation:

```
public class DetailParams
{
    public int ParkId { get; set; }
}

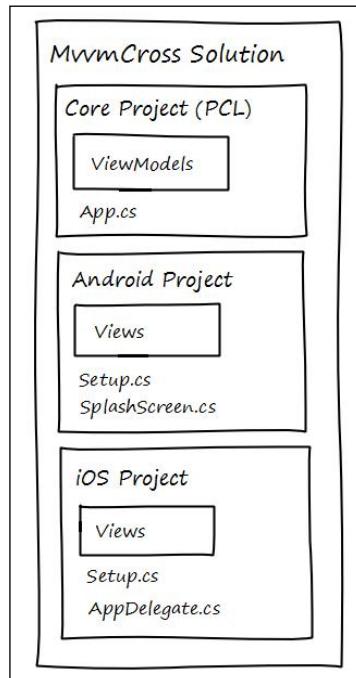
// using the parameters class
ShowViewModel<DetailViewModel>(
    new DetailViewParam() { ParkId = 0 });
```

To receive and use parameters, the destination ViewModel implements an `Init()` method that accepts an instance of the `parameters` class:

```
public class DetailViewModel : MvxViewModel
{
    . .
    public void Init(DetailViewParams parameters)
    {
        // use the parameters here . .
    }
}
```

Solution/project organization

MvvmCross solutions are organized in a way that is similar to how we organized the PCL solution in *Chapter 6, The Sharing Game*. Each MvvmCross solution will have a single core PCL project that houses the reusable code and a series of platform-specific projects that contain the various apps. The following diagram depicts the general structure:



The startup process

MvvmCross apps generally follow a standard startup sequence that is initiated by platform-specific code within each app. There are several classes that collaborate to accomplish the startup; some of these classes reside in the core project and some of them reside in the platform-specific projects. The following sections describe the responsibilities of each of the classes involved.

App.cs

The core project has an `App` class that inherits from `MvxApplication`. The `App` class contains an override to the `Initialize()` method so that at a minimum, it can register the first `ViewModel` that should be presented when the app starts:

```
RegisterAppStart<ViewModels.MasterViewModel>();
```

Setup.cs

Android and iOS projects have a `Setup` class that is responsible for creating the `App` object from the core project during the startup. This is accomplished by overriding the `CreateApp()` method:

```
protected override IMvxApplication CreateApp()
{
    return new Core.App();
}
```

For Android apps, `Setup` inherits from `MvxAndroidSetup`. For iOS apps, `Setup` inherits from `MvxTouchSetup`.

The Android startup

Android apps are kicked off using a special `Activity` splash screen that calls the `Setup` class and initiates the MvvmCross startup process. This is all done automatically for you; all you need to do is include the splash screen definition and make sure it is marked as the launch activity. The definition is as follows:

```
[Activity(
    Label="NationalParks.Droid", MainLauncher = true,
    Icon="@drawable/icon", Theme="@style/Theme.Splash",
    NoHistory=true,
    ScreenOrientation = ScreenOrientation.Portrait)]
public class SplashScreen : MvxSplashScreenActivity
{
    public SplashScreen() :base(Resource.Layout.SplashScreen)
    {
    }
}
```

The iOS startup

The iOS app startup is slightly less automated and is initiated from within the `FinishedLaunching()` method of `AppDelegate`:

```
public override bool FinishedLaunching (
    UIApplication app, NSDictionary options)
{
    _window = new UIWindow (UIScreen.MainScreen.Bounds);

    var setup = new Setup(this, _window);
    setup.Initialize();
    var startup = Mvx.Resolve<IMvxAppStart>();
    startup.Start();

    _window.MakeKeyAndVisible ();
    return true;
}
```

Creating NationalParks.MvvmCross

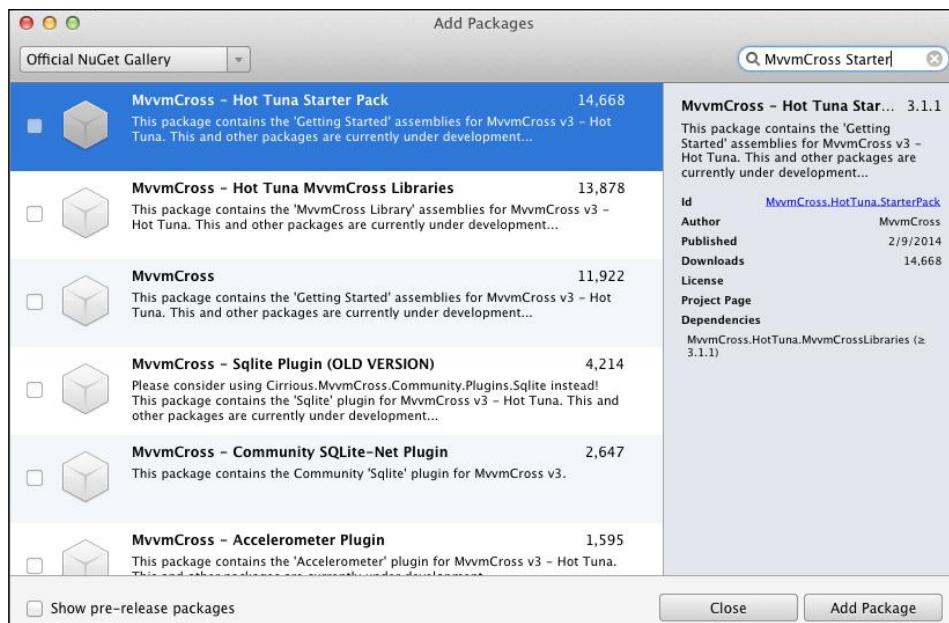
Now that we have basic knowledge of the MvvmCross framework, let's put that knowledge to work and convert the NationalParks app to leverage the capabilities we just learned.

Creating the MvvmCross core project

We will start by creating the core project. This project will contain all the code that will be shared between the iOS and Android app primarily in the form of ViewModels. The core project will be built as a Portable Class Library.

To create NationalParks.Core, perform the following steps:

1. From the main menu, navigate to **File | New Solution**.
2. From the **New Solution** dialog box, navigate to **C# | Portable Library**, enter NationalParks.Core for the project **Name** field, enter NationalParks.MvvmCross for the **Solution** field, and click on **OK**.
3. Add the MvvmCross starter package to the project from NuGet. Select the NationalParks.Core project and navigate to **Project | Add Packages** from the main menu. Enter MvvmCross Starter in the search field.
4. Select the **MvvmCross - Hot Tuna Starter Pack** entry and click on **Add Package**.



5. A number of things were added to `NationalParks.Core` as a result of adding the package, and they are as follows:
 - A `packages.config` file, which contains a list of libraries (`dlls`) associated with the `MvvmCross` starter kit package. These entries are links to actual libraries in the `Packages` folder of the overall solution.
 - A `ViewModels` folder with a sample `ViewModel` named `FirstViewModel`.
 - An `App.cs` class in `App.cs`, which contains an `Initialize()` method that starts the `MvvmCross` app by calling `RegisterAppStart()` to start `FirstViewModel`. We will eventually be changing this to start the `MasterViewModel` class, which will be associated with a `View` that lists national parks.

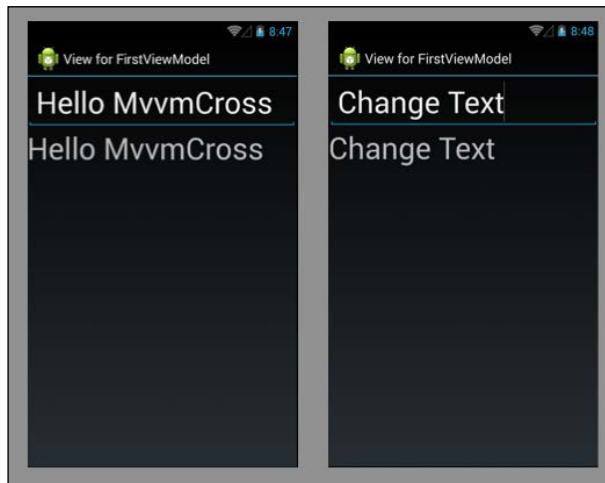
Creating the `MvvmCross` Android app

The next step is to create an Android app project in the same solution.

To create `NationalParks.Droid`, complete the following steps:

1. Select the `NationalParks.MvvmCross` solution, right-click on it, and navigate to **Add | New Project**.
2. From the **New Project** dialog box, navigate to **C# | Android | Android Application**, enter `NationalParks.Droid` for the **Name** field, and click on **OK**.
3. Add the `MvvmCross` starter kit package to the new project by selecting `NationalParks.Droid` and navigating to **Project | Add Packages** from the main menu.
4. A number of things were added to `NationalParks.Droid` as a result of adding the package, which are as follows:
 - `packages.config`: This file contains a list of libraries (`dlls`) associated with the `MvvmCross` starter kit package. These entries are links to an actual library in the `Packages` folder of the overall solution, which contains the actual downloaded libraries.
 - `FirstView`: This class is present in the `Views` folder, which corresponds to `FirstViewModel`, which was created in `NationalParks.Core`.
 - `FirstView`: This layout is present in `Resources\layout`, which is used by the `FirstView` activity. This is a traditional Android layout file with the exception that it contains binding declarations in the `EditText` and `TextView` elements.

- **Setup:** This file inherits from `MvxAndroidSetup`. This class is responsible for creating an instance of the `App` class from the core project, which in turn displays the first ViewModel via a call to `RegisterAppStart()`.
 - **SplashScreen:** This class inherits from `MvxSplashScreenActivity`. The `SplashScreen` class is marked as the main launcher activity and thus initializes the `MvvmCross` app with a call to `Setup`. `Initialize()`.
5. Add a reference to `NationalParks.Core` by selecting the **References** folder, right-click on it, select **Edit References**, select the **Projects** tab, check `NationalParks.Core`, and click on **OK**.
 6. Remove `MainActivity.cs` as it is no longer needed and will create a build error. This is because it is marked as the main launch and so is the new `SplashScreen` class. Also, remove the corresponding `Resources\layout\main.axml` layout file.
 7. Run the app. The app will present `FirstViewModel`, which is linked to the corresponding `FirstView` instance with an `EditView` class, and `TextView` presents the same **Hello MvvmCross** text. As you edit the text in the `EditView` class, the `TextView` class is automatically updated by means of data binding. The following screenshot depicts what you should see:



Reusing NationalParks.PortableData and NationalParks.IO

Before we start creating the Views and ViewModels for our app, we first need to bring in some code from our previous efforts that can be used to maintain parks. For this, we will simply reuse the `NationalParksData` singleton and the `FileHandler` classes that were created previously.

To reuse the `NationalParksData` singleton and `FileHandler` classes, complete the following steps:

1. Copy `NationalParks.PortableData` and `NationalParks.IO` from the solution created in *Chapter 6, The Sharing Game*, to the `NationalParks.MvvmCross` solution folder.
2. Add a reference to `NationalParks.PortableData` in the `NationalParks.Droid` project.
3. Create a folder named `NationalParks.IO` in the `NationalParks.Droid` project and add a link to `FileHandler.cs` from the `NationalParks.IO` project. Recall that the `FileHandler` class cannot be contained in the Portable Class Library because it uses file IO APIs that cannot be references from a Portable Class Library.
4. Compile the project. The project should compile cleanly now.

Implementing the `INotifyPropertyChanged` interface

We will be using data binding to bind UI controls to the `NationalPark` object and thus, we need to implement the `INotifyPropertyChanged` interface. This ensures that changes made to properties of a park are reported to the appropriate UI controls.

To implement `INotifyPropertyChanged`, complete the following steps:

1. Open `NationalPark.cs` in the `NationalParks.PortableData` project.
2. Specify that the `NationalPark` class implements `INotifyPropertyChanged` interface.
3. Select the `INotifyPropertyChanged` interface, right-click on it, navigate to **Refactor | Implement interface**, and press *Enter*. Enter the following code snippet:

```
public class NationalPark : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
        PropertyChanged;
    . . .
}
```

4. Add an `OnPropertyChanged()` method that can be called from each property setter method:

```
void OnPropertyChanged(
    [CallerMemberName] string propertyName = null)
{
    var handler = PropertyChanged;
    if (handler != null)
    {
        handler(this,
            new PropertyChangedEventArgs(propertyName));
    }
}
```

5. Update each property definition to call the setter in the same way as it is depicted for the `Name` property:

```
string _name;
public string Name
{
    get { return _name; }
    set
    {
        if (value.Equals (_name, StringComparison.OrdinalIgnoreCase))
        {
            // Nothing to do - the value hasn't changed;
            return;
        }
        _name = value;
        OnPropertyChanged();
    }
}
```

6. Compile the project. The project should compile cleanly. We are now ready to use the `NationalParksData` singleton in our new project, and it supports data binding.

Implementing the Android user interface

Now, we are ready to create the Views and ViewModels required for our app. The app we are creating will follow the same flow that was used in previous chapters:

- A master list view to view national parks
- A detail view to view details of a specific park
- An edit view to edit a new or previously existing park

The process for creating views and ViewModels in an Android app generally consists of three different steps:

1. Create a ViewModel in the core project with the data and event handlers (commands) required to support the View.
2. Create an Android layout with visual elements and data binding specifications.
3. Create an Android activity, which corresponds to the ViewModel and displays the layout.

In our case, this process will be slightly different because we will reuse some of our previous work, specifically, the layout files and the menu definitions.

To reuse layout files and menu definitions, perform the following steps:

1. Copy `Master.axml`, `Detail.axml`, and `Edit.axml` from the `Resources\layout` folder of the solution created in *Chapter 5, Developing Your First Android App with Xamarin.Android*, to the `Resources\layout` folder in the `NationalParks.Droid` project, and add them to the project by selecting the layout folder and navigating to **Add | Add Files**.
2. Copy `MasterMenu.xml`, `DetailMenu.xml`, and `EditMenu.xml` from the `Resources\menu` folder of the solution created in *Chapter 5, Developing Your First Android App with Xamarin.Android*, to the `Resources\menu` folder in the `NationalParks.Droid` project, and add them to the project by selecting the menu folder and navigating to **Add | Add Files**.

Implementing the master list view

We are now ready to implement the first of our View/ViewModel combinations, which is the master list view.

Creating MasterViewModel

The first step is to create a ViewModel and add a property that will provide data to the list view that displays national parks along with some initialization code.

To create `MasterViewModel`, complete the following steps:

1. Select the `ViewModels` folder in `NationalParks.Core`, right-click on it, and navigate to **Add | New File**.
2. In the **New File** dialog box, navigate to **General | Empty Class**, enter `MasterViewModel` for the **Name** field, and click on **New**.

3. Modify the class definition so that `MasterViewModel` inherits from `MvxViewModel`; you will also need to add a few `using` directives:

```
    . . .
    using Cirrious.CrossCore.Platform;
    using Cirrious.MvvmCross.ViewModels;
    . . .
    namespace NationalParks.Core.ViewModels
    {
        public class MasterViewModel : MvxViewModel
        {
            . . .
        }
    }
```

4. Add a property that is a list of `NationalPark` elements to `MasterViewModel`. This property will later be data-bound to a list view:

```
private List<NationalPark> _parks;
public List<NationalPark> Parks
{
    get { return _parks; }
    set { _parks = value;
          RaisePropertyChanged(() => Parks);
    }
}
```

5. Override the `Start()` method on `MasterViewModel` to load the `_parks` collection with data from the `NationalParksData` singleton. You will need to add a `using` directive for the `NationalParks.PortableData` namespace again:

```
    . . .
    using NationalParks.PortableData;
    . . .
    public async override void Start ()
    {
        base.Start ();
        await NationalParksData.Instance.Load ();
        Parks = new List<NationalPark> (
            NationalParksData.Instance.Parks);
    }
```

6. We now need to modify the app startup sequence so that `MasterViewModel` is the first ViewModel that's started. Open `App.cs` in `NationalParks.Core` and change the call to `RegisterAppStart()` to reference `MasterViewModel`:

```
RegisterAppStart<ViewModels.MasterViewModel>();
```

Updating the Master.axml layout

Update `Master.axml` so that it can leverage the data binding capabilities provided by MvvmCross.

To update `Master.axml`, complete the following steps:

1. Open `Master.axml` and add a namespace definition to the top of the XML to include the `NationalParks.Droid` namespace:

```
xmlns:local="http://schemas.android.com/apk/res/NationalParks.  
Droid"
```

This namespace definition is required in order to allow Android to resolve the MvvmCross-specific elements that will be specified.

2. Change the `ListView` element to a `Mvx.MvxListView` element:

```
<Mvx.MvxListView  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:id="@+id/parksListView" />
```

3. Add a data binding specification to the `MvxListView` element, binding the `ItemsSource` property of the list view to the `Parks` property of `MasterViewModel`, as follows:

```
    . . .  
    android:id="@+id/parksListView"  
    local:MvxBind="ItemsSource Parks" />
```

4. Add a list item template attribute to the element definition. This layout controls the content of each item that will be displayed in the list view:

```
    local:MvxItemTemplate="@layout/nationalparkitem"
```

5. Create the `NationalParkItem` layout and provide `TextView` elements to display both the name and description of a park, as follows:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
    android"  
    xmlns:local="http://schemas.android.com/apk/res/  
        NationalParks.Droid"  
    android:orientation="vertical"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content">  
    <TextView  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:textSize="40sp" />
```

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="20sp" />  
</LinearLayout>
```

6. Add data binding specifications to each of the TextView elements:

```
    . . .  
        local:MvxBind="Text Name" />  
    . . .  
        local:MvxBind="Text Description" />  
    . . .
```



Note that in this case, the context for data binding is an instance of an item in the collection that was bound to `MvxListView`, for this example, an instance of `NationalPark`.



Creating the MasterView activity

Next, create `MasterView`, which is an `MvxActivity` instance that corresponds with `MasterViewModel`.

To create `MasterView`, complete the following steps:

1. Select the `ViewModels` folder in `NationalParks.Core`, right-click on it, navigate to **Add | New File**.
2. In the **New File** dialog, navigate to **Android | Activity**, enter `MasterView` in the **Name** field, and select **New**.
3. Modify the class specification so that it inherits from `MvxActivity`; you will also need to add a few `using` directives as follows:

```
using Cirrious.MvvmCross.Droid.Views;  
using NationalParks.Core.ViewModels;  
.  
.  
.  
namespace NationalParks.Droid.Views  
{  
    [Activity(Label = "Parks")]  
    public class MasterView : MvxActivity  
    {  
        . . .  
    }  
}
```

4. Open `Setup.cs` and add code to initialize the file handler and path for the `NationalParksData` singleton to the `CreateApp()` method, as follows:

```
protected override IMvxApplication CreateApp()
{
    NationalParksData.Instance.FileHandler =
        new FileHandler();
    NationalParksData.Instance.DataDir =
        System.Environment.GetFolderPath(
            System.Environment.SpecialFolder.MyDocuments);
    return new Core.App();
}
```

5. Compile and run the app; you will need to copy the `NationalParks.json` file to the device or emulator using the Android Device Monitor. All the parks in `NationalParks.json` should be displayed.

Implementing the detail view

Now that we have the master list view displaying national parks, we can focus on creating the detail view. We will follow the same steps for the detail view as the ones we just completed for the master view.

Creating DetailViewModel

We start creating `DetailViewModel` by using the following steps:

1. Following the same procedure as the one that was used to create `MasterViewModel`, create a new `ViewModel` named `DetailViewModel` in the `ViewModel` folder of `NationalParks.Core`.
2. Add a `NationalPark` property to support data binding for the view controls, as follows:

```
protected NationalPark _park;
public NationalPark Park
{
    get { return _park; }
    set { _park = value;
          RaisePropertyChanged(() => Park);
    }
}
```

3. Create a `Parameters` class that can be used to pass a park ID for the park that should be displayed. It's convenient to create this class within the class definition of the `ViewModel` that the parameters are for:

```
public class DetailViewModel : MvxViewModel
{
    public class Parameters
    {
        public string ParkId { get; set; }
    }
    ...
}
```

4. Implement an `Init()` method that will accept an instance of the `Parameters` class and get the corresponding national park from `NationalParkData`:

```
public void Init(Parameters parameters)
{
    Park = NationalParksData.Instance.Parks.
        FirstOrDefault(x => x.Id == parameters.ParkId);
}
```

Updating the Detail.axml layout

Next, we will update the layout file. The main changes that need to be made are to add data binding specifications to the layout file.

To update the `Detail.axml` layout, perform the following steps:

1. Open `Detail.axml` and add the project namespace to the XML file:

```
xmlns:local="http://schemas.android.com/apk/res/
    NationalParks.Droid"
```

2. Add data binding specifications to each of the `TextView` elements that correspond to a national park property, as demonstrated for the park name:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/nameTextView"
    local:MvxBind="Text Park.Name" />
```

Creating the DetailView activity

Now, create the `MvxActivity` instance that will work with `DetailViewModel`.

To create `DetailView`, perform the following steps:

1. Following the same procedure as the one that was used to create `MasterView`, create a new view named `DetailView` in the `Views` folder of `NationalParks.Droid`.
2. Implement the `OnCreateOptionsMenu()` and `OnOptionsItemSelected()` methods so that our menus will be accessible. Copy the implementation of these methods from the solution created in *Chapter 6, The Sharing Game*. Comment out the section in `OnOptionsItemSelected()` related to the `Edit` action for now; we will fill that in once the edit view is completed.

Adding navigation

The last step is to add navigation so that when an item is clicked on in `MvxListView` on `MasterView`, the park is displayed in the detail view. We will accomplish this using a `command` property and data binding.

To add navigation, perform the following steps:

1. Open `MasterViewModel` and add an `IMvxCommand` property; this will be used to handle a park that is being selected:

```
protected IMvxCommand ParkSelected { get; protected set; }
```

2. Create an `Action` delegate that will be called when the `ParkSelected` command is executed, as follows:

```
protected void ParkSelectedExec(NationalPark park)
{
    ShowViewModel<DetailViewModel> (
        new DetailViewModel.Parameters ()
        { ParkId = park.Id });
}
```

3. Initialize the `command` property in the constructor of `MasterViewModel`:

```
ParkClicked =
    new MvxCommand<NationalPark> (ParkSelectedExec);
```

4. Now, for the last step, add a data binding specification to `MvvListView` in `Master.axml` to bind the `ItemClick` event to the `ParkClicked` command on `MasterViewModel`, which we just created:

```
local:MvxBind="ItemsSource Parks; ItemClick ParkClicked"
```

5. Compile and run the app. Clicking on a park in the list view should now navigate to the detail view, displaying the selected park.

Implementing the edit view

We are now almost experts at implementing new Views and ViewModels. One last View to go is the edit view.

Creating EditViewModel

Like we did previously, we start with the ViewModel.

To create `EditViewModel`, complete the following steps:

1. Following the same process that was previously used in this chapter to create `EditViewModel`, add a data binding property and create a `Parameters` class for navigation.
2. Implement an `Init()` method that will accept an instance of the `Parameters` class and get the corresponding national park from `NationalParkData` in the case of editing an existing park or create a new instance if the user has chosen the `New` action. Inspect the parameters passed in to determine what the intent is:

```
public void Init(Parameters parameters)
{
    if (string.IsNullOrEmpty (parameters.ParkId))
        Park = new NationalPark ();
    else
        Park =
            NationalParksData.Instance.
            Parks.FirstOrDefault(
                x => x.Id == parameters.ParkId);
}
```

Updating the Edit.axml layout

Update `Edit.axml` to provide data binding specifications.

To update the `Edit.axml` layout, you first need to open `Edit.axml` and add the project namespace to the XML file. Then, add the data binding specifications to each of the `EditView` elements that correspond to a national park property.

Creating the EditView activity

Create a new `MvxActivity` instance named `EditView` to will work with `EditViewModel`.

To create `EditView`, perform the following steps:

1. Following the same procedure as the one that was used to create `DetailView`, create a new View named `EditView` in the `Views` folder of `NationalParks.Droid`.
2. Implement the `OnCreateOptionsMenu()` and `OnOptionsItemSelected()` methods so that the `Done` action will be accessible from the `ActionBar`. You can copy the implementation of these methods from the solution created in *Chapter 6, The Sharing Game*. Change the implementation of `Done` to call the `Done` command on `EditViewModel`.

Adding navigation

Add navigation to two places: when `New (+)` is clicked from `MasterView` and when `Edit` is clicked in `DetailView`. Let's start with `MasterView`.

To add navigation from `MasterViewModel`, complete the following steps:

1. Open `MasterViewModel.cs` and add a `NewParkClicked` command property along with the handler for the command. Be sure to initialize the command in the constructor, as follows:

```
protected IMvxCommand NewParkClicked { get; set; }
protected void NewParkClickedExec()
{
    ShowViewModel<EditViewModel> ();
}
```

Note that we do not pass in a parameter class into `ShowViewModel()`. This will cause a default instance to be created and passed in, which means that `ParkId` will be null. We will use this as a way to determine whether a new park should be created.

2. Now, it's time to hook the `NewParkClicked` command up to the `actionNew` menu item. We do not have a way to accomplish this using data binding, so we will resort to a more traditional approach—we will use the `OnOptionsItemSelected()` method. Add logic to invoke the `Execute()` method on `NewParkClicked`, as follows:

```
case Resource.Id.actionNew:
    ((MasterViewModel)ViewModel).
        NewParkClicked.Execute ();
    return true;
```

To add navigation from `DetailViewModel`, complete the following steps:

1. Open `DetailViewModel.cs` and add a `EditParkClicked` command property along with the handler for the command. Be sure to initialize the command in the constructor, as shown in the following code snippet:

```
protected IMvxCommand EditPark { get; protected set; }
protected void EditParkHandler()
{
    ShowViewModel<EditViewModel> (
        new EditViewModel.Parameters ()
        { ParkId = _park.Id });
}
```

 Note that an instance of the `Parameters` class is created, initialized, and passed into the `ShowViewModel()` method. This instance will in turn be passed into the `Init()` method on `EditViewModel`.

2. Initialize the `command` property in the constructor for `MasterViewModel`, as follows:

```
EditPark =
    new MvxCommand<NationalPark> (EditParkHandler);
```

3. Now, update the `OnOptionsItemSelected()` method in `DetailView` to invoke the `DetailViewModel.EditPark` command when the `Edit` action is selected:

```
case Resource.Id.actionEdit:
    ((DetailViewModel)ViewModel).EditPark.Execute ();
    return true;
```

4. Compile and run `NationalParks.Droid`. You should now have a fully functional app that has the ability to create new parks and edit the existing parks. Changes made to `EditView` should automatically be reflected in `MasterView` and `DetailView`.

Creating the MvvmCross iOS app

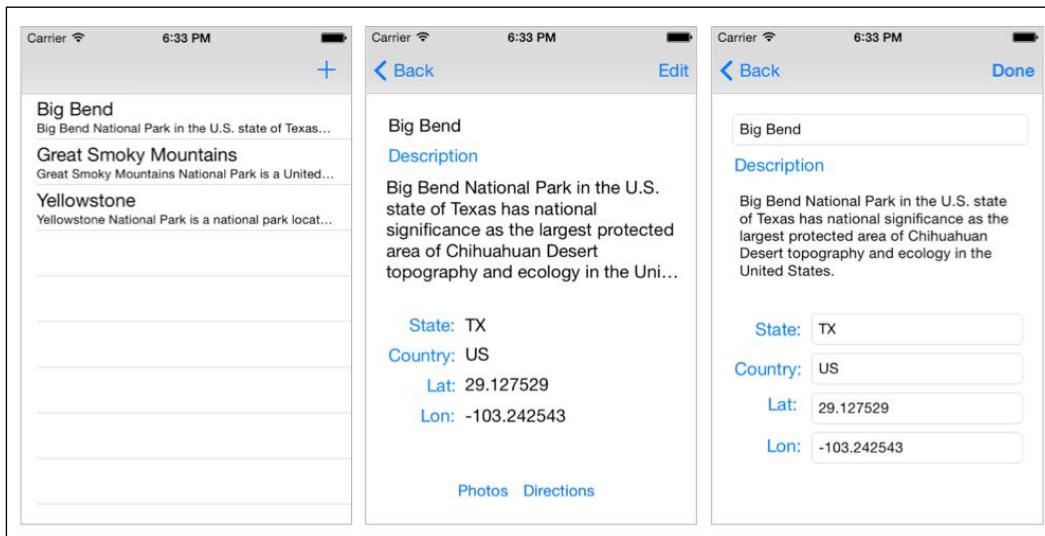
The process of creating the Android app with MvvmCross provides a solid understanding of how the overall architecture works. Creating the iOS solution should be much easier for two reasons: first, we understand how to interact with MvvmCross and second, all the logic we have placed in `NationalParks.Core` is reusable, so that we just need to create the View portion of the app and the startup code.

To create `NationalParks.iOS`, complete the following steps:

1. Select the `NationalParks.MvvmCross` solution, right-click on it, and navigate to **Add | New Project**.
2. From the **New Project** dialog, navigate to **C# | iOS | iPhone | Single View Application**, enter `NationalParks.iOS` in the Name field, and click on **OK**.
3. Add the MvvmCross starter kit package to the new project by selecting `NationalParks.iOS` and navigating to **Project | Add Packages** from the main menu.
4. A number of things were added to `NationalParks.iOS` as a result of adding the package. They are as follows:
 - `packages.config`: This file contains a list of libraries associated with the MvvmCross starter kit package. These entries are links to an actual library in the `Packages` folder of the overall solution, which contains the actual downloaded libraries.
 - `FirstView`: This class is placed in the `Views` folder, which corresponds to the `FirstViewModel` instance created in `NationalParks.Core`.
 - `Setup`: This class inherits from `MvxTouchSetup`. This class is responsible for creating an instance of the `App` class from the core project, which in turn displays the first ViewModel via a call to `RegisterAppStart()`.
 - `AppDelegate.cs.txt`: This class contains the sample startup code, which should be placed in the actual `AppDelegate.cs` file.

Implementing the iOS user interface

We are now ready to create the user interface for the iOS app. The good news is that we already have all the ViewModels implemented, so we can simply reuse them. The bad news is that we cannot easily reuse the storyboards from our previous work; MvvmCross apps generally use XIB files. One of the reasons for this is that storyboards are intended to provide navigation capabilities and an MvvmCross app delegates that responsibility to ViewModel and presenter. It is possible to use storyboards in combination with a custom presenter, but the remainder of this chapter will focus on using XIB files, as this is the more common use. The screen layouts as used in *Chapter 4, Developing Your First iOS App with Xamarin.iOS*, can be used as depicted in the following screenshot:



We are now ready to get started.

Implementing the master view

The first view we will work on is the master view.

To implement the master view, complete the following steps:

1. Create a new ViewController class named `MasterView` by right-clicking on the views folder of `NationalParks.iOS` and navigating to **Add | New File | iOS | iPhone View Controller**.

2. Open `MasterView.xib` and arrange controls as seen in the screen layouts. Add outlets for each of the edit controls.

3. Open `MasterView.cs` and add the following boilerplate logic to deal with constraints on iOS 7, as follows:

```
// ios7 layout
if (RespondsToSelector(new
    Selector("edgesForExtendedLayout")))
    EdgesForExtendedLayout = UIRectEdge.None;
```

4. Within the `ViewDidLoad()` method, add logic to create `MvxStandardTableViewSource` for `parksTableView`:

```
MvxStandardTableViewSource _source;
. . .
_source = new MvxStandardTableViewSource(
    parksTableView,
    UITableViewCellStyle.Subtitle,
    new NSString("cell"),
    "TitleText Name; DetailText Description",
    0);
parksTableView.Source = _source;
```

Note that the example uses the `Subtitle` cell style and binds the national park name and description to the title and subtitle.

5. Add the binding logic to the `ViewDidShow()` method. In the previous step, we provided specifications for properties of `UITableViewCell` to properties in the binding context. In this step, we need to set the binding context for the `Parks` property on `MasterModelView`:

```
var set = this.CreateBindingSet<MasterView,
    MasterViewModel>();
set.Bind (_source).To (vm => vm.Parks);
set.Apply();
```

6. Compile and run the app. All the parks in `NationalParks.json` should be displayed.

Implementing the detail view

Now, implement the detail view using the following steps:

1. Create a new `ViewController` instance named `DetailView`.
2. Open `DetailView.xib` and arrange controls as shown in the following code. Add outlets for each of the edit controls.

3. Open `DetailView.cs` and add the binding logic to the `ViewDidLoad()` method:

```
    this.CreateBinding (this.nameLabel) .
        To ((DetailViewModel vm) => vm.Park.Name) .Apply () ;
    this.CreateBinding (this.descriptionLabel) .
        To ((DetailViewModel vm) => vm.Park.Description) .
        Apply () ;
    this.CreateBinding (this.stateLabel) .
        To ((DetailViewModel vm) => vm.Park.State) .Apply () ;
    this.CreateBinding (this.countryLabel) .
        To ((DetailViewModel vm) => vm.Park.Country) .
        Apply () ;
    this.CreateBinding (this.latLabel) .
        To ((DetailViewModel vm) => vm.Park.Latitude) .
        Apply () ;
    this.CreateBinding (this.lonLabel) .
        To ((DetailViewModel vm) => vm.Park.Longitude) .
        Apply () ;
```

Adding navigation

Add navigation from the master view so that when a park is selected, the detail view is displayed, showing the park.

To add navigation, complete the following steps:

1. Open `MasterView.cs`, create an event handler named `ParkSelected`, and assign it to the `SelectedItemChanged` event on `MvxStandardTableViewSource`, which was created in the `ViewDidLoad()` method:

```
    . . .
    _source.SelectedItemChanged += ParkSelected;
    . . .
protected void ParkSelected(object sender, EventArgs e)
{
    . . .
}
```

2. Within the event handler, invoke the `ParkSelected` command on `MasterViewModel`, passing in the selected park:

```
( (MasterViewModel)ViewModel).ParkSelected.Execute (
    (NationalPark)_source.SelectedItem);
```

3. Compile and run `NationalParks.iOS`. Selecting a park in the list view should now navigate you to the detail view, displaying the selected park.

Implementing the edit view

We now need to implement the last of the Views for the iOS app, which is the edit view.

To implement the edit view, complete the following steps:

1. Create a new `ViewController` instance named `EditView`.
2. Open `EditView.xib` and arrange controls as in the layout screenshots. Add outlets for each of the edit controls.
3. Open `EditView.cs` and add the data binding logic to the `ViewDidLoad()` method. You should use the same approach to data binding as the approach used for the details view.
4. Add an event handler named `DoneClicked`, and within the event handler, invoke the `Done` command on `EditViewModel`:

```
protected void DoneClicked (object sender, EventArgs e)
{
    ((EditViewModel)ViewModel).Done.Execute();
}
```

5. In `ViewDidLoad()`, add `UIBarButtonItem` to `NavigationItem` for `EditView`, and assign the `DoneClicked` event handler to it, as follows:

```
NavigationItem.SetRightBarButtonItem(
    new UIBarButtonItem(UIBarButtonItemSystemItem.Done,
    DoneClicked), true);
```

Adding navigation

Add navigation to two places: when **New (+)** is clicked from the master view and when **Edit** is clicked on in the detail view. Let's start with the master view.

To add navigation to the master view, perform the following steps:

1. Open `MasterView.cs` and add an event handler named `NewParkClicked`. In the event handler, invoke the `NewParkClicked` command on `MasterViewModel`:

```
protected void NewParkClicked(object sender,
    EventArgs e)
{
    ((MasterViewModel)ViewModel).
        NewParkClicked.Execute ();
}
```

2. In `ViewDidLoad()`, add `UIBarButtonItem` to `NavigationItem` for `MasterView` and assign the `NewParkClicked` event handler to it:

```
NavigationItem.SetRightBarButtonItem(  
    new UIBarButtonItem(UIBarButtonItemSystemItem.Add,  
        NewParkClicked), true);
```

To add navigation to the details view, perform the following steps:

1. Open `DetailView.cs` and add an event handler named `EditParkClicked`. In the event handler, invoke the `EditParkClicked` command on `DetailViewModel`:

```
protected void EditParkClicked (object sender,  
    EventArgs e)  
{  
    ((DetailViewModel)ViewModel).EditPark.Execute ();  
}
```

2. In `ViewDidLoad()`, add `UIBarButtonItem` to `NavigationItem` for `MasterView`, and assign the `EditParkClicked` event handler to it:

```
NavigationItem.SetRightBarButtonItem(  
    new UIBarButtonItem(UIBarButtonItemSystemItem.Edit,  
        EditParkClicked), true);
```

Refreshing the master view list

One last detail that needs to be taken care of is to refresh the `UITableView` control on `MasterView` when items have been changed on `EditView`.

To refresh the master view list, perform the following steps:

1. Open `MasterView.cs` and call `ReloadData()` on `parksTableView` within the `ViewDidAppear()` method of `MasterView`:

```
public override void ViewDidAppear (bool animated)  
{  
    base.ViewDidAppear (animated);  
    parksTableView.ReloadData();  
}
```

2. Compile and run `NationalParks.iOS`. You should now have a fully functional app that has the ability to create new parks and edit existing parks. Changes made to `EditView` should automatically be reflected in `MasterView` and `DetailView`.

Considering the pros and cons

After completing our work, we now have the basis to make some fundamental observations. Let's start with the pros:

- MvvmCross definitely increases the amount of code that can be reused across platforms. The ViewModels house the data required by the View, the logic required to obtain and transform the data in preparation for viewing, and the logic triggered by user interactions in the form of commands. In our sample app, the ViewModels were somewhat simple; however, the more complex the app, the more reuse will likely be gained.
- As MvvmCross relies on the use of each platform's native UI frameworks, each app has a native look and feel and we have a natural layer that implements platform-specific logic when required.
- The data binding capabilities of MvvmCross also eliminate a great deal of tedious code that would otherwise have to be written.

All of these positives are not necessarily free; let's look at some cons:

- The first con is complexity; you have to learn another framework on top of Xamarin, Android, and iOS.
- In some ways, MvvmCross forces you to align the way your apps work across platforms to achieve the most reuse. As the presentation logic is contained in the ViewModels, the views are coerced into aligning with them. The more your UI deviates across platforms; the less likely it will be that you can actually reuse ViewModels.

With these things in mind, I would definitely consider using MvvmCross for a cross-platform mobile project. Yes, you need to learn an addition framework and yes, you will likely have to align the way some of the apps are laid out, but I think MvvmCross provides enough value and flexibility to make these issues workable. I'm a big fan of reuse and MvvmCross definitely pushes reuse to the next level.

Summary

In this chapter, we reviewed the high-level concepts of MvvmCross and worked through a practical exercise in order to convert the national parks apps to use the MvvmCross framework and the increase code reuse. In the next chapter, we will follow a similar approach to exploring the Xamarin.Forms framework in order to evaluate how its use can affect code reuse.

8

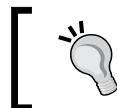
Sharing with Xamarin.Forms

In this chapter, we will discuss Xamarin.Forms, a cross-platform development framework. With this in mind, we will cover the following areas:

- Pages, Views (Controls), and Layouts
- Navigation in Xamarin.Forms
- XAML and code-behind classes
- Data binding
- Renderers
- The DependencyService API
- App startup
- Project organization
- Converting the NationalParks app to use Xamarin.Forms

An insight into the Xamarin.Forms framework

The Xamarin.Forms framework can be used to develop mobile apps for Android, iOS, and Windows Phone. It uses virtually the same source code base for each platform while still providing a platform-specific look and feel. Xamarin.Forms is available for use from any of the paid licenses available at Xamarin or from the 30-day evaluation.



While we mention that Xamarin.Forms apps can run on Windows Phone, the licensing, configuration, and development details for Windows Phone are beyond the scope of this book.

Unlike the approaches described previously in this book, Xamarin.Forms provides you with a set of abstractions that cover the entire user interface, thus allowing the UI code and specification to be reused across multiple platforms. At runtime, Xamarin.Forms renders user interfaces using Controls that are native to each platform, which allows apps to retain a native look and feel.

This chapter is divided into two main sections: in the first section, we cover the core concepts that need to be understood prior to using Xamarin.Forms, and in the second section, we will convert our NationalParks app to use the Xamarin.Forms framework.

Pages

A **Page** is a visual element that organizes the content a user sees on the screen at a single time. A Xamarin.Forms Page is essentially similar to an Android activity or an iOS View controller. Xamarin.Forms provides the following base Pages for use in your apps, where you can find a description accompanied with each type:

Type	Description
ContentPage	This allows you to organize a set of Controls, or Views, into a Layout for display and interaction with the user
MasterDetailPage	This manages two pages – a master and a detail page – and the navigation between them
NavigationPage	This manages navigation over a set of other pages
TabbedPage	This manages a set of child pages and allows you to navigate via tabs
CarouselPage	This manages a set of child pages and allows you to navigate via swipe

Views

A **View** is a visual control (or widget) that presents information and allows the user to interact with your app (things such as buttons, labels, and edit boxes). These controls generally inherit properties from the `View` class. The following table represents the list of Views provided by Xamarin.Forms at the time of writing this book:

ActivityIndicator	BoxView	Button	DatePicker
Editor	Entry	Image	Label
ListView	OpenGLView	Picker	ProgressBar
SearchBar	Slider	Stepper	Switch
TableView	TimePicker	WebView	

Layouts

Controls are hosted within a special type of View called a **Layout**. There are two different types of Layouts: managed and unmanaged. Managed Layouts are responsible for arranging their hosted Controls, and unmanaged Layouts require the developer to specify how controls should be arranged. Xamarin.Forms provides the following Layouts:

Layout	Description
ContentView	This is a Layout that can contain child views. Generally, ContentView is not used directly, but is used as a base for other layouts.
Frame	This is a Layout that can contain a single child view and provide framing options such as padding.
ScrollView	This Layout is capable of scrolling its child views.
AbsoluteLayout	This Layout allows its child views to be positioned by absolute positions as requested by the app.
Grid	This Layout allows content to be displayed in rows and columns.
RelativeLayout	This Layout positions views relative to other views it owns by use of constraints.
StackLayout	This Layout positions views horizontally or vertically in a single line.

Cells

A **Cell** is a special type of Control used to arrange information in a list; specifically, `ListView` or `TableView`. Cells derive from the `Element` class rather than the `VisualElement` class and act as a template to create `VisualElements`.

Xamarin.Forms provides the following types of Cells:

Cell type	Description
EntryCell	This is a Cell with a label and single text entry field.
SwitchCell	This is a Cell with a label and switch view (on/off).
TextCell	This is a Cell with primary and secondary text. Generally, the primary text is used as a title and the secondary text as a subtitle.
ImageCell	This is a TextCell that also includes an image.

Navigation

Navigation in a Xamarin.Forms app is accomplished with the use of the navigation property of `VisualElement`. This is generally accessed via a `Page`. The navigation property is typed as the `INavigation` interface, which provides the following methods:

Type	Description
<code>PushAsync()</code>	This method pushes a <code>Page</code> on the navigation stack
<code>PushModalAsync()</code>	This method pushes a <code>Page</code> on the navigation stack as a modal dialog
<code>PopAsync()</code>	This method pops the current <code>Page</code> off the navigation stack
<code>PopModalAsync()</code>	This method pops the current modal <code>Page</code> off the navigation stack
<code>PopToRootAsync()</code>	This method pops all the <code>Pages</code> off the navigation stack, except the root <code>Page</code>

The beauty of navigation in Xamarin.Forms lies in its simplicity. To navigate to a new `Page` and pass data into the new `Page`, all you need to do is create an instance of the new `Page` passing the data in the constructor and then push this `Page` on the navigation stack, as demonstrated by the following sample code snippet:

```
public partial class ParkDetailPage : ContentPage
{
    . .
    public void EditClicked(object sender, EventArgs e)
    {
        Navigation.PushModalAsync (
            new ParkEditPage (_park));
    }
}
```

Defining Xamarin.Forms user interfaces

Like many UI frameworks, Xamarin.Forms allows two different approaches to create user interfaces: declarative and programmatic:

- **Programmatic approach:** When using this approach, the developers embed API calls to construct a UI, and control the size and placement
- **Declarative approach:** When using this approach, the developers create XAML files that define the content and layout for a user interface

Extensible Application Markup Language (XAML)

Extensible Application Markup Language (XAML) is an XML-based language developed by Microsoft. XAML allows developers to use XML to specify a hierarchy of objects to instantiate. It can be used in a number of ways, but most successfully as a means to specify user's interfaces in **Windows Presentation Foundation (WPF)**, Silverlight, Windows Runtime, and now Xamarin.Forms.

XAML files are parsed at build time to verify objects that have been specified and at runtime to instantiate the hierarchy of objects.

In addition to specifying a hierarchy of objects, XAML also allows developers to specify property values and assign event handlers. However, it does not allow you to embed code or logic.

The following XAML file defines the content for a ContentPage view:

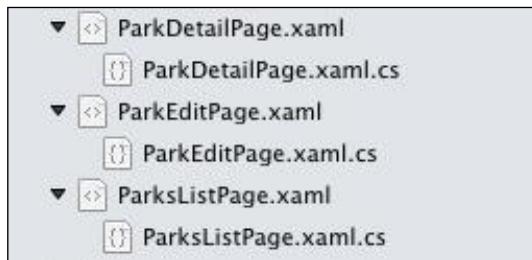
```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/...
    xmlns:x="http://schemas.microsoft.com/winfx...
    x:Class="NationalParks.ParkEditPage">
    <StackLayout Orientation="Vertical"
        HorizontalOptions="StartAndExpand">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label
                Text="Name :"
                Grid.Row="0" Grid.Column="0" />
            <Entry
                x:Name="descriptionEntry"
                Text="{Binding Name}"
                Grid.Row="0" Grid.Column="1"
                HorizontalOptions="FillAndExpand" />
            . .
        </Grid>
        <Button
            x:Name="doneButton"
            Text="Done"
            Clicked="DoneClicked" />
    </StackLayout>
</ContentPage>
```

Take a look at the previous XAML specification:

- The class name is `ParkEditPage` and is specified in the `ContentPage` element
- A `Grid` Layout is used to organize the content in the Page
- Two components are assigned property names, `nameEntry` and `doneButton`
- The `doneButton` component is assigned a `Clicked` event handler named `DoneClicked`

Code-behind classes

When you create a Page in a Xamarin.Forms app, two files are actually created: an XAML file and a class file. Xamarin Studio nests the class files under the XAML files in the **Solution** pad, as shown in the following screenshot:



The `.xaml.cs` files are sometimes referred to as code-behind classes. They are created to contain all the app logic in event handlers that go hand in hand with the Page definition. The following example shows the code-behind class for `ParkEditPage`:

```
public partial class ParkEditPage : ContentPage
{
    NationalPark _park;
    public ParkEditPage()
    {
        InitializeComponent();
    }
    protected void DoneClicked(object sender, EventArgs e)
    {
        // perform event handler logic
    }
}
```

You should take note of the following aspects of the class definition:

- `ParkEditPage` is a partial class definition.
- The `DoneClicked()` event handler is defined within this class file. This is the event handler that was assigned to the `Done` button in XAML.
- There are no property definitions defined in the earlier file.

So, where are the property definitions? Xamarin Studio generates a second code file each time the app is built. For our example, the file will be named `ParkEditPage.xaml.g.cs` and will contain the following code snippet:

```
public partial class ParkEditPage : ContentPage {
    private Entry nameEntry;
    . .
    private Button doneButton;
    private void InitializeComponent() {
        this.LoadFromXaml(typeof(ParkEditPage));
        nameEntry =
            this.FindByName <Entry>("nameEntry");
        . .
        doneButton =
            this.FindByName <Button>("doneButton");
    }
}
```

You should take note of the following points here:

- There are two properties defined on the `ParkEditPage` file: `nameEntry` and `doneButton`. These are generated directly from the names found in the XAML file.
- A method named `InitializeComponent()` is generated. This method must be called from any constructors defined in `ParkEditPage.xaml.cs`.
- The `InitializeComponent()` method calls `LoadFromXaml()` to instantiate all the objects defined by `ParkEditPage.xaml`.
- The `InitializeComponent()` method calls `FindByName()` to bind each property to its corresponding instance.

Data binding

The concepts behind data binding are covered in detail in *Chapter 7, Sharing with MvvmCross*, under the section titled *Data binding*. Xamarin.Forms provides a data binding capability that follows the same architecture as MvvmCross, **Windows Presentation Foundation (WPF)**, Silverlight, and Windows Runtime.

Within a Xamarin.Forms app, binding specifications are generally specified in XAML. The following XAML specification demonstrates binding the `Text` property of an `Entry` control to the `Name` property of a `NationalPark` object:

```
<Entry x:Name="nameEntry"
      Text="{Binding Name}"
      Grid.Row="0" Grid.Column="1"
      HorizontalOptions="FillAndExpand" />
```

Generally, the binding context is set with code. The following example demonstrates how to programmatically set the binding context at a Page level to a `NationalPark` object:

```
public ParkEditPage (NationalPark park)
{
    InitializeComponent ();
    _park = park;
    BindingContext = _park;
}
```

In the previous example, the binding context was set for an entire Page. Sometimes, Controls provide a binding context that needs to be set to accomplish data binding. The following example demonstrates setting the binding context for a `ListView` control:

```
parksListView.ItemsSource =
    NationalParksData.Instance.Parks;
```



Note that the binding context for the `ListView` control is a property named `ItemsSource`.

Using Renderers

Xamarin.Forms uses platform-native controls to render user interfaces that allow apps to maintain a look and feel that the users would expect for each platform. This is accomplished with the use of Renderers. Pages, Layouts, and Controls represent the set of abstractions used to describe a user interface. Each of these elements is rendered using a Renderer class, which in turn creates a native control based on the platform the app is running on.

Developers can create their own Renderers in order to customize the way a particular Control is rendered on a platform.

Native features and the DependencyService API

Until now, we have primarily focused on working with abstractions that can be reused across all platforms. What if you need access to platform-specific capabilities? That's where the `DependencyService` API comes in. The `DependencyService` API is an API that allows each platform to register a platform-specific service that can be called by shared code through a common interface.

Using the `DependencyService` API involves the following three steps:

1. Firstly, you need to create an interface that exposes the platform-specific methods that must be implemented for each platform the app will run on.
2. After this step, create an implementation of the interface for each platform and register the implementation using an `assembly` attribute.
3. To conclude, call `DependencyService.Get<MyInterface>` from the shared code to look up the appropriate implementation and invoke services on the returned instance.

We will demonstrate the use of the `DependencyService` API later in this chapter in the section titled *Adding calls to DependencyService*.

App startup

Xamarin.Forms apps start up as native apps, meaning the traditional startup sequence is followed. During the startup sequence, an app performs the following two tasks:

1. Make a call to initialize the Xamarin.Forms runtime.
2. Start the first Page.

Shared App classes

By default, Xamarin.Forms apps have a shared `App` class created, which contains a single static method that returns the first Page that should be presented when an app starts up. The following code demonstrates this:

```
public class App
{
    public static Page Get MainPage()
    {
        return new HelloWorldPage();
    }
}
```

This simple approach allows the platform-specific startup code in each app to call the `Get MainPage()` method in order to determine which Page to start with. Therefore, it is only specified at one place.

iOS apps

In a Xamarin.Forms iOS app, initialization is performed in the `FinishedLaunching()` method of the `AppDelegate` class, as shown in the following sample demonstration:

```
[Register("AppDelegate")]
public partial class AppDelegate : UIApplicationDelegate
{
    UIWindow window;
    public override bool FinishedLaunching(UIApplication app,
    NSDictionary options)
    {
        Forms.Init();
        window = new UIWindow(UIScreen.MainScreen.Bounds);
        window.RootViewController =
            App.Get MainPage().CreateViewController();
```

```
        window.MakeKeyAndVisible();
        return true;
    }
}
```

Android apps

In a Xamarin.Forms Android app, initialization is done in the `Activity` instance marked with the `MainLauncher=true` attribute. This is shown in the following sample code snippet:

```
namespace HelloWorld.Android
{
    [Activity(Label="HelloWorld", MainLauncher=true)]
    public class MainActivity : AndroidActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            Xamarin.Forms.Forms.Init(this, bundle);
            SetPage(App.Get MainPage());
        }
    }
}
```

Project organization

Xamarin.Forms projects are generally created using one of the following two project templates, which can be found by navigating to **C# | Mobile Apps** of the **New Solution** dialog box:

- **Blank App (Xamarin.Forms Potable)**
- **Blank App (Xamarin.Forms Shared)**

The difference between these two templates is the type of project created to house the shared code. Using the first template, shared code is housed in a Portable Class Library, and using the second template, shared code is housed in a shared project. Shared projects allow all referencing projects to reuse the code it contains, but the code is compiled specifically for each referencing project.



If you plan to add a Windows Phone project to the solution at some point, you will be well versed to go with the PCL solution. It will require you to work within the restrictions of a PCL, but will ensure your code is compatible with more platforms.

After creating a Xamarin.Forms solution, you will see that three actual projects were created. The first project contains the shared code, the second project contains the iOS code, and the third project contains the Android code. If we are successful with Xamarin.Forms, the bulk of the code will end up in the shared project. The following screenshot shows an example of a project created with the PCL template:



Creating the NationalParks Xamarin.Forms app

Now that we have a solid understanding of Xamarin.Forms, let's convert our NationalParks app to use the new framework. For this exercise, we will follow the same app flow that we have used in the iOS app so far, meaning that we will have a list page, a detail page to view, and an edit page to add and update.

Creating the solution

We will start by creating an entirely new project by performing the following steps:

1. To start with, access the **File** menu and navigate to **New | New Solution**.
2. In the **New Solution** dialog box, navigate to **C# | Mobile Apps**, select the **Blank App (Xamarin.Forms Portable)** template, enter **NationalParks** in the **Name** field, choose the appropriate **Location** value, and click on **OK**.

3. Review the project structure. You will see the following pointers:
 - Open `AppDelegate.cs` in the `NationalParks.iOS` project. Note the calls to `Forms.Init()` and `App.GetMainPage()`.
 - Open `MainActivity.cs` in the `NationalParks.Android` project. Note the calls to `Forms.Init()` and `App.GetMainPage()`.
 - Open `App.cs` in the `NationalParks` project. Note the static method, `Get MainPage()`.
4. To finish, run the `NationalParks.Android` and `NationalParks.iOS` projects.

Adding NationalParks.PortableData

Our next step is to bring in the storage solution from *Chapter 7, Sharing with MvvmCross*. Perform the following steps to add the storage solution to our new `Xamarin.Forms` solutions:

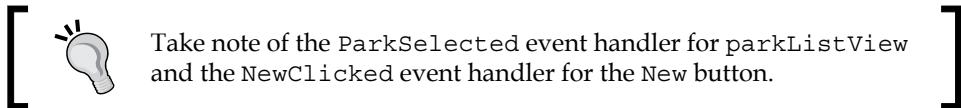
1. Firstly, you need to copy the `NationalParks.PortableData` and `NationalParks.IO` projects from the solution folder of *Chapter 7, Sharing with MvvmCross*, to the new solution folder.
2. Add each project to the new solution folder by selecting the solution, right-clicking on it, navigating to **Add | Add Existing Project**, and selecting the project file, for example, `NationalParks.IO.csproj`.
3. Add the `NationalParks.PortableData` project to the new `NationalParks`, `NationalParks.Android`, and `NationalParks.iOS` projects as a reference by selecting the `References` folder in each of the projects, right-clicking on them, choosing **Edit References**, and selecting `NationalParks.PortableData`.
4. We now need to add a link to the `FileHandler.cs` file to both the `NationalParks.Android` and `NationalParks.iOS` projects. For each project, create a new folder named `NationalParks.IO` and add a link to `FileHandler.cs` by selecting the new folder, right-clicking on it, navigating to **Add | Add Files**, selecting `FileHandler`, choosing **Open**, selecting **Add a link to the file**, and clicking on **OK**.
5. To verify all of the previous steps, you should compile the new solution.

Implementing ParksListPage

We can now begin work on the user interface starting with a list view to display the parks by performing the following steps:

1. Select the NationalParks project, right-click on it, and navigate to **Add | New File**. From the **New File** dialog box, navigate to **Forms | Forms ContentPage Xaml**, enter ParksListPage for the **Name** field, and choose **New**.
2. You should now open ParkListPage.xaml. You will see an empty ContentPage element. Add StackLayout, which is vertically oriented, with a child ListView and Button instances, as follows:

```
<StackLayout Orientation="Vertical"
    HorizontalOptions="StartAndExpand">
    <ListView x:Name="parksListView"
        IsVisible="true"
        ItemSelected="ParkSelected">
        </ListView>
        <Button Text="New"
            Clicked="NewClicked" />
    </StackLayout>
```



Take note of the ParkSelected event handler for parkListView and the NewClicked event handler for the New button.

3. Now, let's add the row definitions for ListView. The ListView element has a DataTemplate property that defines a layout for each row. The following Layout should define a label for the name and description of the park. This should be placed within the ListView element of the XAML:

```
<ListView.ItemTemplate>
    <DataTemplate>
        <ViewCell>
            <ViewCell.View>
                <StackLayout Orientation="Vertical"
                    HorizontalOptions="StartAndExpand">
                    <Label Text="{Binding Name}"
                        HorizontalOptions="FillAndExpand" />
                </StackLayout>
            </ViewCell.View>
        </ViewCell>
    </DataTemplate>
</ListView.ItemTemplate>
```

```
    </ViewCell.View>
    </ViewCell>
  </DataTemplate>
</ListView.ItemTemplate>
```

Note the binding specifications for the two Label views.

4. Open `App.cs` in the `NationalParks` project and change the main page to `ParksListPage`. We also need to create `NavigationPage` as the owner of `ParksListPage` to support push and pop navigation. The `Get MainPage()` method should contain the following code:

```
public static Page Get MainPage ()
{
    NavigationPage mainPage =
        new NavigationPage(new ParksListPage());
    return mainPage;
}
```

5. Open `AppDelegate.cs` in the `NationalParks.iOS` project. You should then add the following initialization of code to the `FinishedLaunching()` method just before the `Forms.Init()` call:

```
// Initialize data service
NationalParksData.Instance.DataDir =
    Environment.CurrentDirectory;
NationalParksData.Instance.FileHandler =
    new FileHandler();
```

6. Open `MainActivity.cs` in the `NationalParks.Android` project. Once you're in, add the following initialization code to the `OnCreate()` method just before the call to `Forms.Init()`:

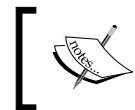
```
// Initialize data service
NationalParksData.Instance.DataDir =
    System.Environment.GetFolderPath(
        System.Environment.SpecialFolder.MyDocuments);
NationalParksData.Instance.FileHandler =
    new FileHandler();
```

7. Open `ParksListPage.xaml.cs`, add a method to load the parks data, and set the binding context:

```
public async Task LoadData()
{
    await NationalParksData.Instance.Load();
    parksListView.ItemsSource =
        NationalParksData.Instance.Parks;
}
```

8. Add a call to `LoadData()` from the constructor:

```
InitializeComponent();  
LoadData();
```



You will not be able to use `await` on the `LoadData()` method because it's being called from a constructor. In this case, there is actually no need to await the call.

9. The last step is to create two stub event handlers for `NewClicked` and `ParkSelected`, which we will fill in later as we complete the app.
10. We are now ready to test our work. Compile and run both the `NationalParks.iOS` and `NationalParks.Android` apps.

Implementing ParkDetailPage

Now, we need a page to display the details of a Park. To create `ParkDetailPage`, perform the following steps:

1. Add a new `ContentPage` instance named `ParkDetailPage`.
2. For `ParkDetailPage`, we will display a series of `Label` views in `Grid` and a set of `Buttons` below `Grid` to initiate actions. All this content will be hosted within `StackLayout`, which is vertically oriented. Start by adding `StackLayout` like we did in the previous section.
3. Add a `Grid` layout with a series of the `Label` view to display the properties of `NationalPark`, as follows:

```
<Grid>  
    <Grid.ColumnDefinitions>  
        <ColumnDefinition Width="Auto" />  
        <ColumnDefinition Width="*" />  
    </Grid.ColumnDefinitions>  
    <Label Text="Name:"  
          Grid.Row="0" Grid.Column="0" />  
    <Label Text="{Binding Name}"  
          Grid.Row="0" Grid.Column="1"  
          HorizontalOptions="FillAndExpand" />  
    . . .  
</Grid>
```



Note the `Grid.Row` and `Grid.Column` specifications, which control how the `Label` and `Entry` views are positioned.

- Now, add three Button definitions for the actions that can be taken from the Page, as follows:

```
<Button Text="Edit"
        WidthRequest="175"
        HorizontalOptions="Center"
        Clicked="EditClicked" />
<Button Text="Directions"
        WidthRequest="175"
        HorizontalOptions="Center"
        Clicked="DirectionsClicked" />
<Button Text="Photos"
        WidthRequest="175"
        HorizontalOptions="Center"
        Clicked="PhotosClicked" />
```

- Add a constructor that accepts a NationalPark instance to be displayed. The following code sample demonstrates what is needed:

```
NationalPark _park;
public ParkDetailPage (NationalPark park)
{
    InitializeComponent ();
    _park = park;
    BindingContext = _park;
}
```



Note that the last line in the constructor sets BindingContext. This tells the Page how to resolve the binding specifications declared in XAML.

- Add stub event handlers for EditClicked, DirectionsClicked, and PhotosClicked.
- Now, we need to return to the ParksListPage class and add the navigation logic. Open ParksListPage.xaml.cs and update the ParkSelected() event handler to make a call to PushAsync() for ParkDetailPage, as follows:

```
protected void ParkSelected(object sender,
SelectedEventArgs e)
{
    Navigation.PushAsync (new
    ParkDetailPage ((NationalPark) e.SelectedItem));
}
```

- Compile and run both the NationalParks.iOS and NationalParks.Android apps.

Using DependencyService to show directions and photos

As we discussed earlier, the `DependencyService` API allows apps to take advantage of platform-specific features. We will demonstrate the use of `DependencyService` to implement the ability to show directions and photos for a park.

Creating the interface

The first step is to create an interface in the shared project that describes the methods that need to be supported. To create the `IParkInfoServices` interface, perform the following steps:

1. To begin with, select the `NationalParks` project, right-click on it, and navigate to **Add | New File**.
2. Navigate to **General | Empty Interface**, enter `IParkInfoServices` in the **Name** field, and choose **New**.
3. You now need to create two methods on the interface, one to show directions and one to show photos; each should accept `NationalPark` as a parameter:

```
public interface IParkInfoServices
{
    void ShowDirections(NationalPark park);
    void ShowPhotos(NationalPark park);
}
```

Creating the iOS implementation

Now, let's create an iOS implementation by performing the following steps:

1. Select the `NationalParks.iOS` project, right-click on it, and navigate to **Add | New File**. In this dialog box, navigate to **General | Empty Class** in the **New File** dialog box, enter `iOSParkInfoServices` in the **Name** field, and choose **New**.
2. Add using clauses for the namespaces, `Xamarin.Forms`, `NationalParks`, `NationalParks.PortableData`, and `NationalParks.iOS`.
3. Change the `iOSParkInfoServices` class specification so that it implements `IParkInfoServices`.
4. Select `IParkInfoService`, right-click on it, navigate to **Refactor | Implement interface**, and press *Enter*.

5. You should then provide implementations for the two methods calls, as follows:

```
public void ShowDirections(NationalPark park)
{
    if ((park.Latitude.HasValue) &&
        (park.Longitude.HasValue))
    {
        NSUrl url = new NSUrl (
            String.Format (
                "http://maps.apple.com/maps?daddr={0},{1}&saddr=Current
                Location", park.Latitude, park.Longitude));
        UIApplication.SharedApplication.OpenUrl (url);
    }
}
public void ShowPhotos(NationalPark park)
{
    UIApplication.SharedApplication.OpenUrl (
        new NSUrl(String.Format (
            "http://www.bing.com/images/search?q={0}",
            park.Name)));
}
```

6. Finally, add the following `Dependency` attribute to the `class` file outside the namespace definition, as follows:

```
[assembly: Dependency (typeof (iOSParkInfoServices))]
namespace NationalParks.iOS
{
```

The `Dependency` attribute registers the class with `DependencyService` so that when `Get()` is called, a platform-specific implementation can be located.

Creating the Android implementation

Now, let's create an Android implementation by performing the following steps:

1. Select the `NationalParks.Android` project, right-click on it, and navigate to **Add | New File**. You should then navigate to **General | Empty Class** in the **New File** dialog box, enter `AndroidParkInfoServices` in the **Name** field, and choose **New**.
2. Add using clauses for the namespaces, `Xamarin.Forms`, `NationalParks`, `NationalParks.PortableData`, and `NationalParks.Droid`.

3. Change the `AndroidParkInfoServices` class specification so that it implements `IParkInfoServices`.
4. After this, select `IParkInfoService`, right-click on it, navigate to **Refactor | Implement interface**, and press *Enter*.
5. Provide implementations for the two method calls, as follows:

```
public void ShowDirections(NationalPark park)
{
    if ((park.Latitude.HasValue) &&
        (park.Longitude.HasValue))
    {
        Intent mapIntent = new Intent (Intent.ActionView,
            Android.Net.Uri.Parse (
                String.Format ("geo:0,0?q={0},{1}&z=16 ({2})",
                    park.Latitude, park.Longitude,
                    park.Name)));
        Forms.Context.StartActivity (mapIntent);
    }
}
public void ShowPhotos(NationalPark park)
{
    Intent urlIntent = new Intent (Intent.ActionView);
    urlIntent.SetData (Android.Net.Uri.Parse (
        String.Format (
            "http://www.bing.com/images/search?q={0}",
            park.Name)));
    Forms.Context.StartActivity (urlIntent);
}
```



Note the use of `Forms.Context`. In the case of Android, this contains the currently executing `Activity`; in our case, `MainActivity`.

6. Add the following `Dependency` attribute to the `class` file outside the namespace definition, as follows:

```
[assembly: Dependency (typeof (AndroidParkInfoServices))]
namespace NationalParks.iOS
{
    . . .
```

Adding calls to DependencyService

Now, we need to add code to the shared project in order to actually invoke `ShowDirections()` and `ShowPhotos()`. All you need to do is open `ParkDetailPage.xaml.cs` and fill in the stub implementations for `DirectionsClicked()` and `PhotosClicked()`, as follows:

```
public void DirectionsClicked(
    object sender, EventArgs e)
{
    DependencyService.Get<IParkInfoServices> () .
        ShowDirections (_park);
}
public void PhotosClicked(
    object sender, EventArgs e)
{
    DependencyService.Get<IParkInfoServices> () .
        ShowPhotos (_park);
}
```

Running the app

We are finally ready to run the app. While there were several steps, the `DependencyService` API provided a very clean approach to separate shared and platform-specific code. Run both the `NationalParks.iOS` and `NationalParks.Android` apps.

Implementing ParkEditPage

Now, we need a Page to update the park information. To implement `ParkEditPage`, perform the following steps:

1. To begin with, add a new ContentPage named `ParkDetailsPage`.
2. We will use a similar Layout for `ParkEditPage` as we did for `ParkDetailPage` with the exception that we will use `Entry` views to allow editing of the properties of `NationalPark`. Add a `StackLayout` and `Grid` instance to `ParkEditPage`, and add a series of `Label` and `Entry` views for each property of `NationalPark`, as shown in the following code snippet:

```
<Label Text="Name:" Grid.Row="0" Grid.Column="0" />
<Entry x:Name="nameEntry" Text="{Binding Name}" Grid.Row="0" Grid.Column="1" HorizontalOptions="FillAndExpand" />
```

3. You can then add a Done button to complete the editing process, as follows:

```
<Button x:Name="doneButton"
        Text="Done"
        WidthRequest="175"
        HorizontalOptions="Center"
        Clicked="DoneClicked" />
```

4. Create two constructors, one that accepts a NationalPark instance and will be used to edit existing parks, and one that does not accept a NationalPark instance and will be used to create a new park, as shown in the following code snippet:

```
public ParkEditPage()
{
    InitializeComponent();
    _park = new NationalPark();
    BindingContext = _park;
}
public ParkEditPage (NationalPark park)
{
    InitializeComponent();
    _park = park;
    BindingContext = _park;
}
```

5. Create the DoneClicked() event handler with a call to save the updated park, and a navigation call to PopAsync() to return to the Page that displayed ParkEditPage, as follows:

```
protected void DoneClicked(object sender, EventArgs e)
{
    NationalParksData.Instance.Save(_park);
    Navigation.PopAsync();
}
```

6. We now need to add navigation logic to both ParkListPage and ParkDetailPage. Open ParkDetailPage.xaml.cs and fill in the EditClicked() event handler with a call to PushAsync() in order to display ParkEditPage. Pass the park that is being viewed to the ParkEditPage constructor, as follows:

```
public void EditClicked(object sender, EventArgs e)
{
    Navigation.PushAsync(new ParkEditPage(_park));
}
```

7. Open `ParkListPage.xaml.cs` and fill in the `NewClicked()` event handler with a call to `PushAsync()` in order to display `ParkEditPage`. Call the empty `ParkEditPage` constructor so that a new park will be created, as follows:

```
protected void NewClicked(object sender, EventArgs e)
{
    Navigation.PushAsync(new ParkEditPage());
}
```

8. We are now ready with our app; compile and run both the `NationalParks.iOS` and `NationalParks.Android` apps.

Considering the pros and cons

As we have seen from the exercise, Xamarin.Forms provides a solid approach to dramatically increase the amount of code reused across your mobile apps; it has many great features:

- XAML is a great way to define user interfaces and allows you to create properties and assign event handlers in a convenient, concise way
- The data binding capabilities are great and eliminate a lot of tedious mind-numbing code from being written
- The `DependencyService` API provides a great way to access platform-specific capabilities
- The Renderer architecture provides for ultimate customizability

However, at the time of writing this book, Xamarin.Forms is still somewhat immature, and there are some weaknesses:

- There is no visual designer for the XAML code, so you have to construct your UI and run the app to see it visually rendered
- Due to the newness of the framework, there is a limited number of examples available for reference, and many of the examples use code to construct the UI rather than XAML
- Validation capabilities seem pretty weak

These criticisms should not be taken too strongly; cross-platform UI frameworks are tough to build, and I feel confident that Xamarin is on the right track and will evolve the framework rapidly.

Summary

In this chapter, we reviewed the capabilities of Xamarin.Forms and converted our existing NationalParks app to use the framework. In the next chapter, we will look at the process of preparing an iOS app for distribution.

9

Preparing Xamarin.iOS Apps for Distribution

In this chapter, we will discuss activities related to preparing a Xamarin.iOS app for distribution and look at the various options for distributing apps. While many of the activities we will discuss are an integral part of any iOS app deployment, we will try and narrow the scope of our coverage to aspects that are unique to developing an app with Xamarin.iOS. We will cover the following topics:

- App profiling
- iOS Build settings for distributing apps
- App distribution options

Preparing for distribution

At this point, our app is built and functioning the way we want; most of the work is done. We now turn our attention to preparing our app for distribution. This section discusses the following three aspects of preparing an app for distribution:

- **App profiling:** Here we will be looking at memory allocation issues and performance bottlenecks
- **iOS Application settings:** Here we will be updating informational settings such as version and build numbers
- **iOS Build settings:** Here we will be adjusting settings that affect the code being generated based on target devices, desired performance characteristics, and deployable size

Profiling Xamarin.iOS apps

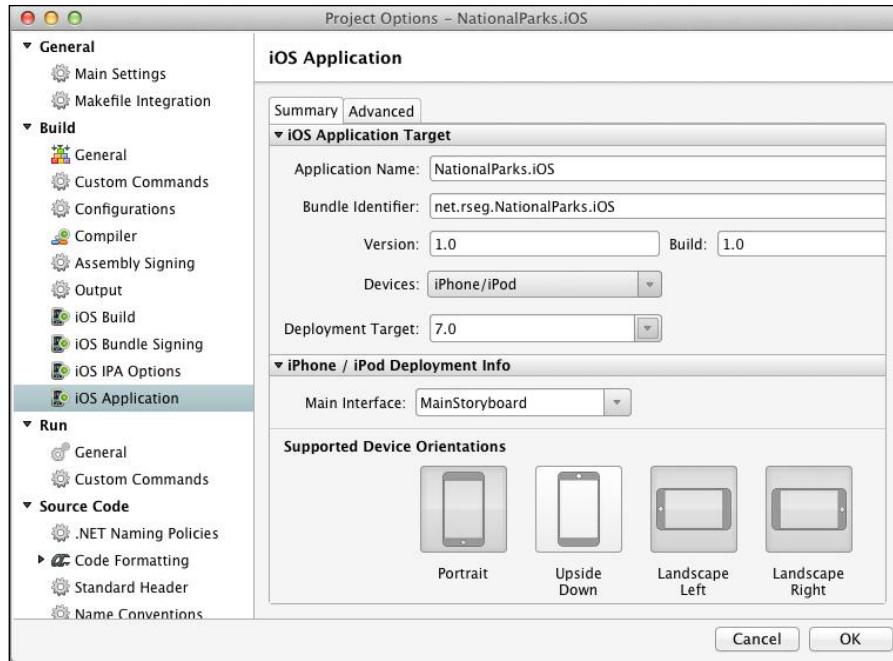
Profiling allows developers to monitor their apps during execution and identify issues related to memory allocation and performance bottlenecks. The activity of profiling can be performed throughout the life cycle of developing an app, but it is especially beneficial to incorporate profiling into the latter stages of the process as a final verification prior to distribution.

Xamarin.iOS developers have two tools to choose from for profiling apps: MonoTouch Profiler and Apple's Instruments app. We will not replicate the existing documentation for these apps but simply provide the following links for reference:

Tool	URL
MonoTouch Profiler	http://docs.xamarin.com/guides/ios/deployment,_testing,_and_metrics/monotouch_profiler/
Apple's Instruments app	http://docs.xamarin.com/guides/ios/deployment,_testing,_and_metrics/walkthrough_Apples_instrument/

iOS Application (Info.plist) settings

It's likely that most of the settings you need to make in `Info.plist` will have already been made by the time you are ready to start the distribution process. However, there are a few settings you likely need to update, specifically, the version and build settings. The following screenshot shows the **iOS Application** settings screen:



iOS Build settings

Xamarin.iOS provides numerous options to optimize the build process based on the devices that are being targeted, the size of the deployable app, and the execution speed. The following sections discuss the most important settings related to producing a final build for distribution.

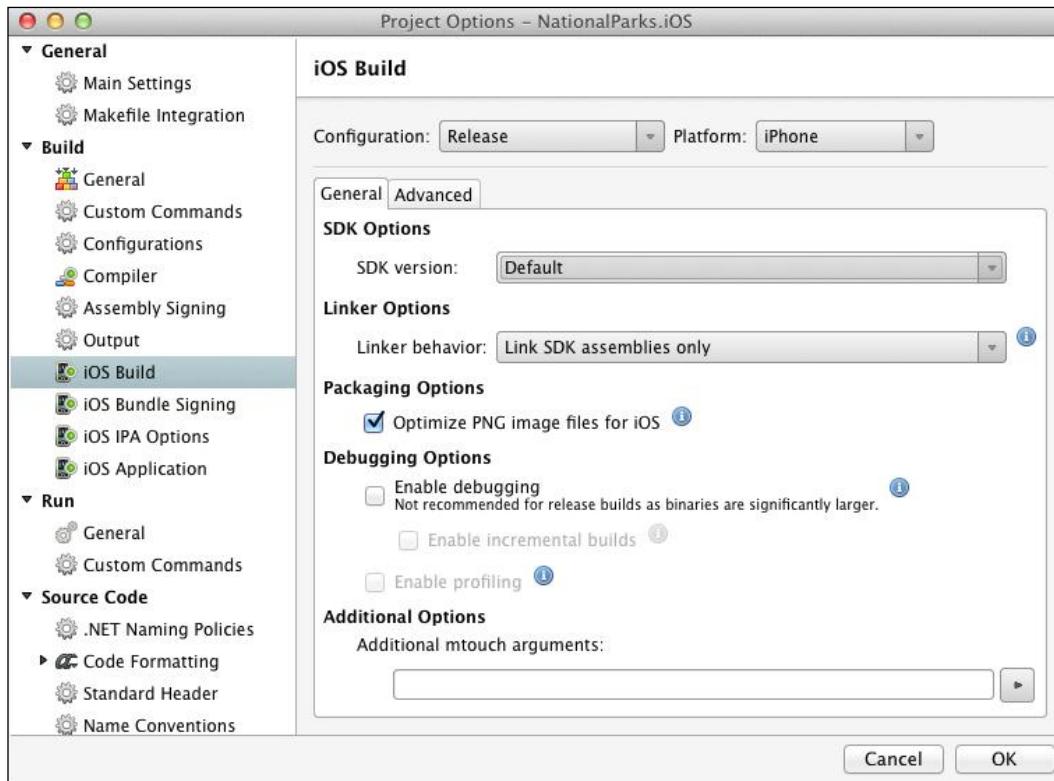
SDK Options

The SDK version should be set to the minimum iOS version that the app can be deployed to. It's likely that this setting would have already been established during the development process.

Linker Options

The mTouch tool used to build Xamarin.iOS apps includes a linker, where the aim of the linker is to reduce the size of the resulting app. The linker accomplishes this by performing static analysis on the code in your app, evaluating which classes and methods in the referenced assemblies are actually used, and removing classes, methods, and properties that are not used.

Options for the linker can be set in **Project Options | iOS Build** under the **General** tab, as shown in the following screenshot:



The following options can be set to control the linking process:

- **Don't link:** This option disables the linker and ensures that all referenced assemblies are included without modification. You should note that this is the default setting for builds that target the iOS simulator because excluding the time-consuming static analysis process saves time. From this, the resulting large DLLs can still be deployed relatively quickly to the simulator.

- **Link SDK assemblies only:** This option tells the linker to operate on only the SDK assemblies (which are the assemblies that ship with Xamarin.iOS). This is the default setting for builds that target a device.
- **Link all assemblies:** This option tells the linker to operate on the entire app as well as on all referenced assemblies. This allows the linker to use a larger set of optimizations and results in the smallest possible application. However, when the linker runs in this mode, there is a greater chance that it will break portions of your code due to false assumptions made by the static analysis process. In particular, static analysis can get tripped up through usage of reflection, serialization, or any code where a type or member instance is not statically referenced.

The following table summarizes the results of linking the two versions of the NationalParks app produced in *Chapter 6, The Sharing Game*:

	File linking version	PCL version
Don't link	47.5 MB	48.4 MB
Link SDK assemblies only	6.7 MB	7.3 MB
Link all assemblies	5.8 MB	6.4 MB

As you can see from the table, the biggest difference in application size is achieved when going from **Don't link** to **Link SDK assemblies only**.

Overriding the linker

The linker provides great benefits as demonstrated in the previous section. However, there might be times when you need to override the default behavior of the linker as the linker might remove type and member instances that are actually used by your app. This will result in runtime exceptions relating to these types and/or member not being found. The following table describes three ways to alter the behavior of the linker in order to avoid losing important types and members:

Technique	Description
Preserving code	If you determine from testing that the linker is removing classes or methods needed by your app, you can explicitly tell the linker to always include them by using the <code>Preserve</code> attribute on a class and/or method. To preserve the entire type use: <code>[Preserve (AllMembers = true)]</code> To preserve a single member use: <code>[Preserve (Conditional=true)]</code>

Technique	Description
Skipping assemblies	In some cases, you might need to tell the linker to skip entire assemblies because you do not have the ability to modify the source code (third-party libraries). This can be accomplished by using the command line option <code>linkskip</code> . For example: <code>--linkskip=someassembly</code>
Disable Link Away	One optimization the linker employs is to remove code that is very unlikely to be used on an actual device; those features that are marked as unsupported or disallowed. On rare occasions, these features might be needed for your app to function. This optimization can be disabled by using the command line option <code>-nolinkaway</code> .

Debugging options

Debugging options should always be disabled for release builds. Enabling debugging can result in significantly larger binaries.

Code generation options

Code generation options control the code being created during the build process based on the processor(s) being targeted and the performance characteristics desired. The options we have under this setting are explained in the following sections.

Supported architectures

Supported architectures identify the processor architectures that should be supported by the resulting build. The original iPhone through to the iPhone 3G, used an ARMv6 CPU. Newer models of iPhone and iPad use either the ARMv7 or ARMv7s architecture while the iPhone 5s introduced the use of A7 processor based on the ARMv8a architecture.

ARMv6 has not supported Xcode versions prior to Xcode 4.5. If you need to build for older devices, you will need to use an earlier version of Xcode installed.

LLVM optimizing compiler

Xamarin.iOS comes with two different code generation engines: the normal Mono code generation engine and the one based on the LLVM optimizing compiler. The LLVM engine produces both faster and smaller code than the Mono engine at the cost of compile time. Thus, the Mono code generation engine is convenient to use as you develop an app, whereas the LLVM engine is preferred for builds that will be distributed.

ARM thumb instruction set

The ARM thumb instruction set is a more compact instruction set used by ARM processors. By enabling the Thumb support, you can reduce the size of your executable at the expense of slower execution times. Thumb is supported by ARMv6, ARMv7, and ARMv7s.

Distributing Xamarin.iOS apps

Xamarin.iOS supports all of the traditional distribution methods that iOS developers have access to. There is a great deal of information about distribution of iOS apps on the Xamarin website and the Apple developer website. We make no attempt to replicate those comprehensive repositories. The following sections are intended to provide a general overview from a Xamarin.iOS perspective.

The Ad Hoc and enterprise distributions

The Ad Hoc distribution and enterprise distributions allow an app to be distributed without going through the App Store. Ad Hoc is generally used to support testing efforts leading up to a general release. Enterprise is used to distribute apps that are not intended for the general public, but are instead intended for use by users within a single enterprise. In either case, an iOS App Store Package (IPA) must be created.



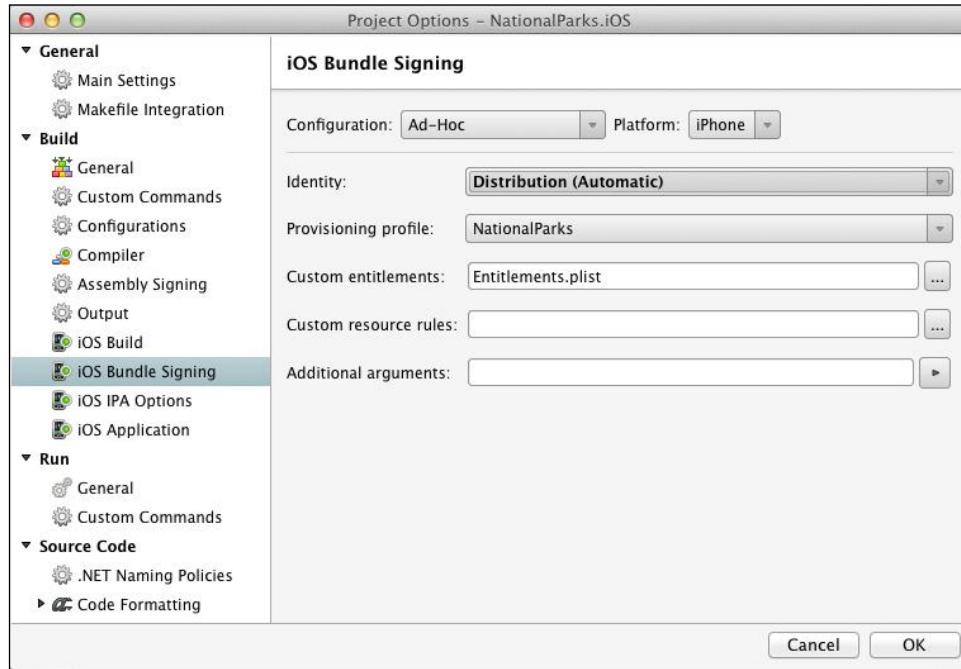
Producing an enterprise distribution requires an Enterprise account from Apple and an Enterprise Xamarin.iOS license.



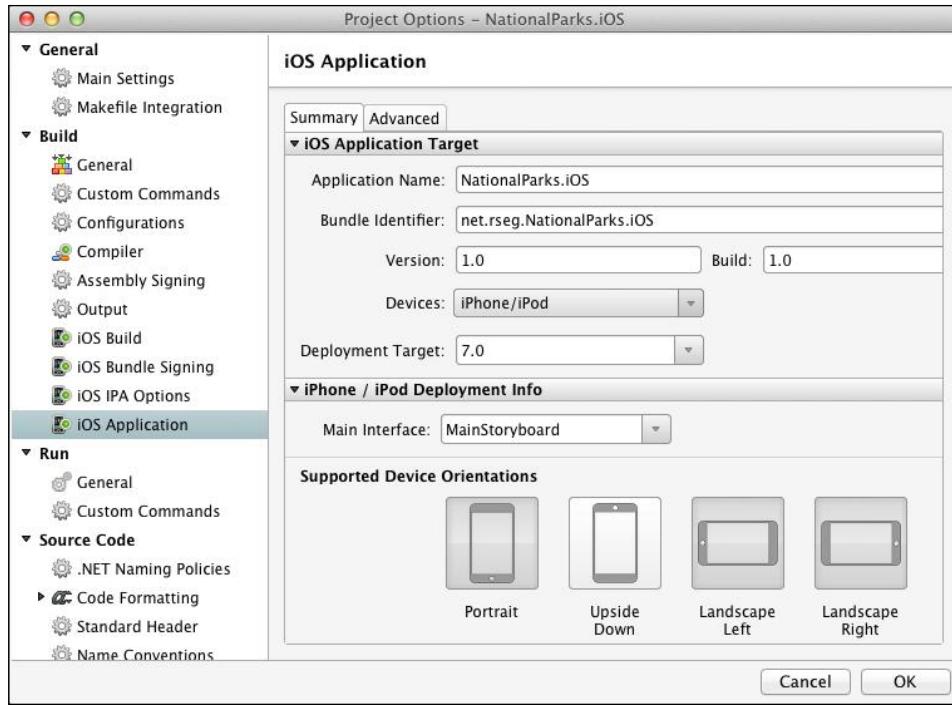
To create an IPA, we will perform the following steps:

1. Create and install a distribution profile for your app on developer.apple.com. Detailed instructions for this procedure can be found in the section titled *Creating and Installing a Distribution Profile* at http://docs.xamarin.com/guides/ios/deployment,_testing,_and_metrics/app_distribution_overview/publishing_to_the_app_store/.

2. Set the **Provisioning profile** option to be used for the build of the newly installed profile by navigating to **Project Options** | **Build** | **iOS Bundle Signing**, as shown in the following screenshot:



3. Set the **Bundle Identifier** option on the app to the same value that was used while creating the distribution profile by navigating to **Project Options** | **Build** | **iOS Application**, as shown in the following screenshot:



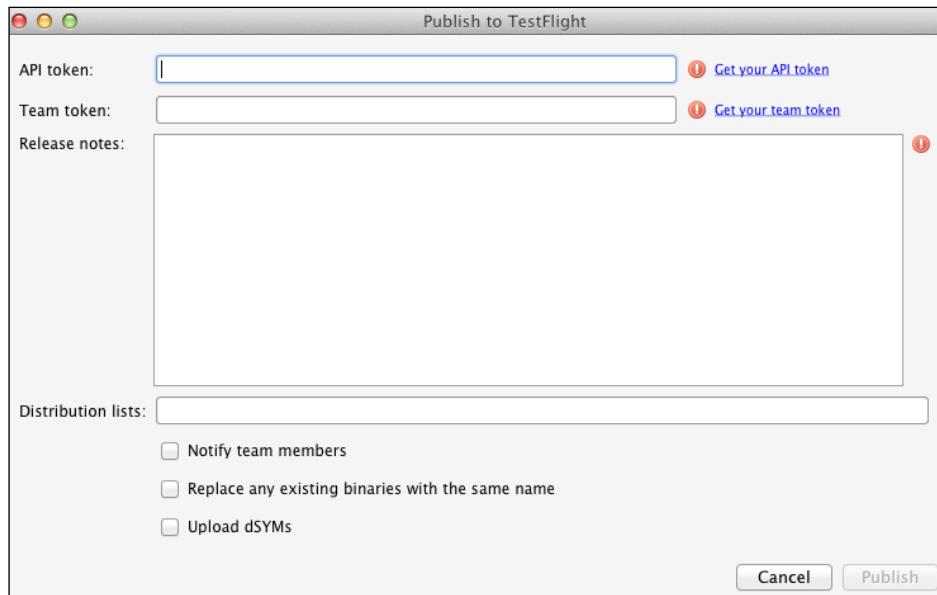
- Once the IPA has been created, simply navigate to the IPA in finder, double-click on it, and it will be opened in iTunes. iTunes can now be used to sync the app on devices.

TestFlight distribution

TestFlight is a cloud-based app distribution service used to distribute pre-released versions of your app. Xamarin Studio provides its integration with the **TestFlight** service so that **Ad Hoc** builds can be uploaded to **TestFlight** directly from within the IDE. Prior to uploading a build, you must establish an account and define the testing team(s) and app within the **TestFlight** service. This can be accomplished by <https://www.testflightapp.com>.

To upload a build to TestFlight, perform the following steps:

1. Select **Ad Hoc** for the build type and navigate from **Project | Publish** to **TestFlight** from the main menu.
2. Enter the **API token** and **Team token** values assigned by **TestFlight** when you set up your app and team. You can click on the link next to these fields to display the appropriate value in a browser.



3. Enter **Release notes** to let the testers know what has been fixed and/or added in the new release.
4. Enter **Distribution lists** and turn on **Notify team members** to have an e-mail notification sent out with the release notes.
5. Select the options, **Replace existing binaries with the same name** and **Upload dSYMs**, and click on **Publish**. Xamarin Studio will build the app and upload it to TestFlight.

App Store submission

Distributing apps through the App Store is much the same for Xamarin.iOS apps as any other iOS app. With the exception of producing a release build, most of the work is done outside of Xamarin Studio. You do need to enter the **Provisioning Profile** value in the **iOS Bundle Signing** section of the **Project Options** dialog box.

The following link provides detailed steps to publish a Xamarin.iOS app to the App Store: http://docs.xamarin.com/guides/ios/deployment,_testing,_and_metrics/app_distribution_overview/publishing_to_the_app_store/.

Summary

In this chapter, we discussed the activities related to preparing an app for distribution, the distribution channels available, and the processes involved in distributing an app. In the next chapter, we will look at the same aspects of distributing a Xamarin.Android app.

10

Preparing Xamarin.Android Apps for Distribution

In this chapter, we will discuss activities related to preparing a Xamarin.Android app for distribution and look at the various options for distributing apps. Many of the activities we will discuss are an integral part of any Android app deployment. However, in this chapter, we will try to narrow the scope of our coverage to aspects that are unique to developing with Xamarin.Android. We will cover the following topics:

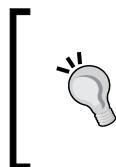
- App profiling
- Android Build settings for distributing apps
- App distribution options

Preparing for a release APK

Prior to publishing a signed APK file for release, there are a number of activities that need to be completed. The following sections discuss topics that should be considered prior to producing a release APK.

Profiling Xamarin.Android apps

The Xamarin.Android Business license provides limited support to profile Android apps. Profiling can be a very effective way to identify memory leaks and process bottlenecks.



We will not cover profiling within this book, but the following link provides an overview of using the profiling capabilities of Xamarin.Android: http://docs.xamarin.com/guides/android/deployment,_testing,_and_metrics/profiling.

In addition to using the tools provided with Xamarin.Android, traditional Android profiling tools such as Traceview and `dmtracedump` can be used. You can find more information at <http://developer.android.com/tools/debugging/debugging-tracing.html>.

Disabling debug

When developing a Xamarin.Android app, Xamarin Studio supports debugging with the use of **Java Debug Wire Protocol (JDWP)**. This feature needs to be disabled in release builds as it poses a security risk. You have two different ways to accomplish this:

- Changing the settings in `AndroidManifest.xml`
- Changing the settings in `AssemblyInfo.cs`

Changing the settings in `AndroidManifest.xml`

The first method can be done by using the following listing, which shows you how to turn off JDWP debugging from the `AndroidManifest.xml` file:

```
<application . . .
    android:debuggable="false" . . .
</application>
```

Changing the settings in `AssemblyInfo.cs`

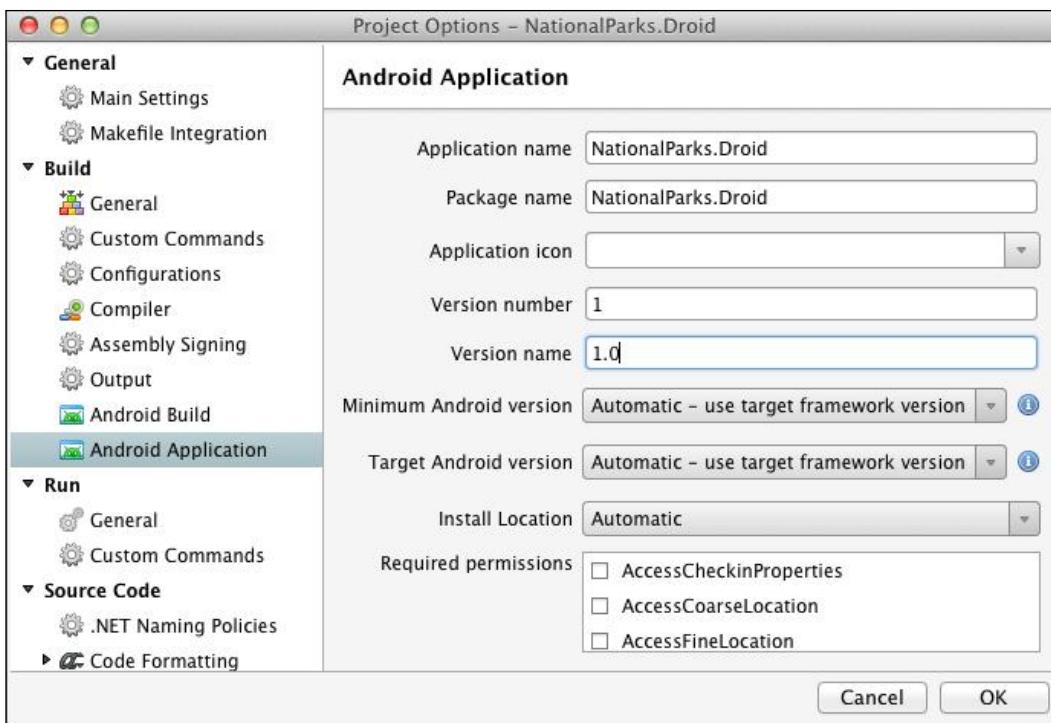
The alternative method is done by disabling JDWP through `AssemblyInfo.cs`. This has the advantage of being based on the currently selected configuration. The following listing shows how to use a conditional directive to turn JDWP debugging off:

```
#if RELEASE
[assembly: Application(Debuggable=false)]
```

```
#else
[assembly: Application(Debuggable=true)]
#endif
```

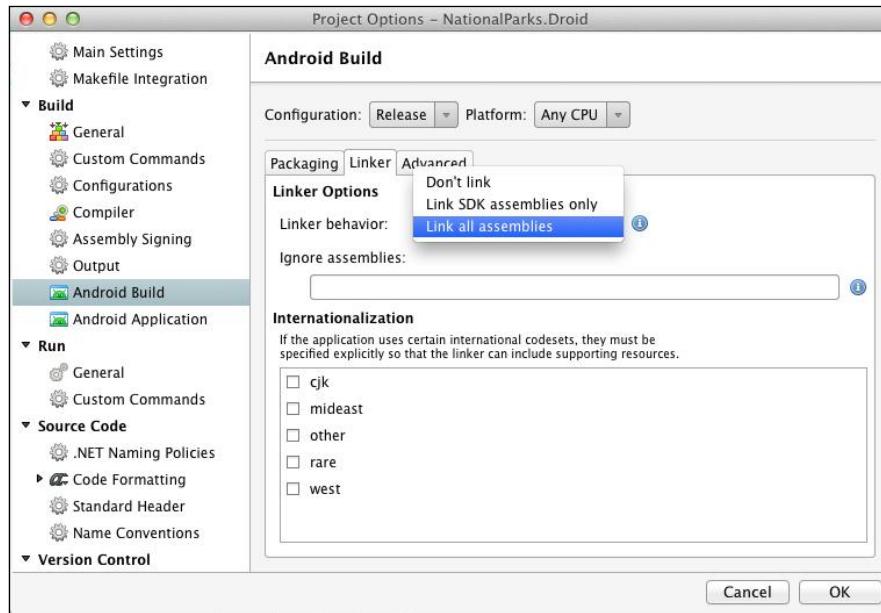
Android Application (AndroidManifest.xml) settings

By the time you start considering deployment of your app, you will have already established most of the settings needed in `AndroidManifest.xml`. However, you will need to update the version information. Keep in mind, the version number is used by the Android platform during the installation process to determine whether an APK is an update to an existing app. A version name is a free-form text and can be used to track app versions in any manner desired. This can be accomplished by opening the **Project Options** dialog box and navigating to **Build | Android Application**, or by double-clicking on `NationalParks/Properties/AndroidManifest.xml`. The following screenshot depicts the **Android Application** settings dialog box:



Linker Options

During a build, Xamarin.Android performs static analysis of the assemblies that make up an app and attempts to eliminate type and member instances that are not needed. The settings that control this process can be viewed and set in the **Project Options** dialog box under the **Android Build** section, as shown in the following screenshot:



Xamarin.Android supports the same linker options as Xamarin.iOS. When viewing and adjusting the **Linker Options** settings, be sure to first select **Release** from the **Configuration** drop-down box. The following linking options are available:

- **Don't link:** This option disables the linker and ensures that all referenced assemblies are included without modification. This is the default setting for builds that target the iOS Simulator. This eliminates the time-consuming process of linking, and deploying a large file to the simulator is relatively quick.
- **Link SDK assemblies only:** This option tells the linker to operate only on the SDK assemblies; those assemblies that ship with Xamarin.iOS. This is the default setting for builds that target a device.

- **Link all assemblies:** This option tells the linker to operate on the entire app, as well as all referenced assemblies. This allows the linker to use a larger set of optimizations and results in the smallest possible application. However, when the linker runs in this mode, there is a greater chance that it might break portions of your code due to false assumptions made by the static analysis process. In particular, the use of reflection and serialization can trip up the static analysis.

The following table shows how the APK file size varies based on the linker setting:

	File linking version	PCL version
Don't link	26.4 MB	27.5 MB
Link SDK assemblies only	4.3 MB	4.3 MB
Link all assemblies	4.1 MB	4.2 MB

Overriding the linker

In some cases, the linking option can have negative side effects, such as important types and/or members being accidentally eliminated. It is important for an application that has been compiled and linked in release mode to be thoroughly tested before distribution in order to identify such issues. In some situations, you should conduct tests beyond the initial developer's testing, and this should be conducted using an APK file produced in release mode.

In the event that you encounter any runtime exceptions related to missing types or trouble locating specific methods, you can make use of one of the following methods to give explicit instructions to the linker.

Preserving code with attributes

If you determine from testing that the linking is removing classes or methods needed by your app, you can explicitly tell the linker to always include them by using the `Preserve` attribute on a class and/or method.

To preserve the entire type, use the following command:

```
[Preserve (AllMembers = true)]
```

To preserve a single member, use the following command:

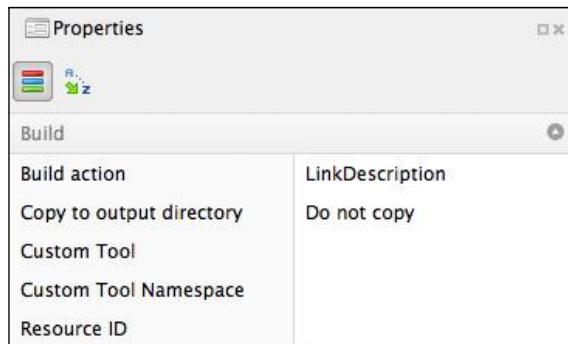
```
[Preserve (Conditional=true)]
```

Preserving code with custom linker files

There are times when you might not have access to the source code, but still need to preserve specific types and/or members. This can be accomplished with a custom linker file. The following example instructs the linker to always include specific members for a type:

```
<?xml version="1.0" encoding="UTF-8" ?>
<linker>
    <assembly fullname="Mono.Android">
        <type fullname="Android.Widget.AdapterView">
            <method name="GetGetAdapterHandler"/>
            <method name="GetSetAdapter_Landroid
                _widget_Adapter_Handler"/>
        </type>
    </assembly>
</linker>
```

You can add a custom-linking file to a project by adding a simple XML file and populating it with similar content to the previous example; it does not really matter where you place it in the project structure. After adding the file to the project, select the file, open the **Properties** pad, and choose **LinkDescription** for **Build action**, as shown in the following screenshot:



Skipping assemblies

You can also instruct the linker to skip entire assemblies so that all the types and members will be retained. This can be accomplished in the following two ways:

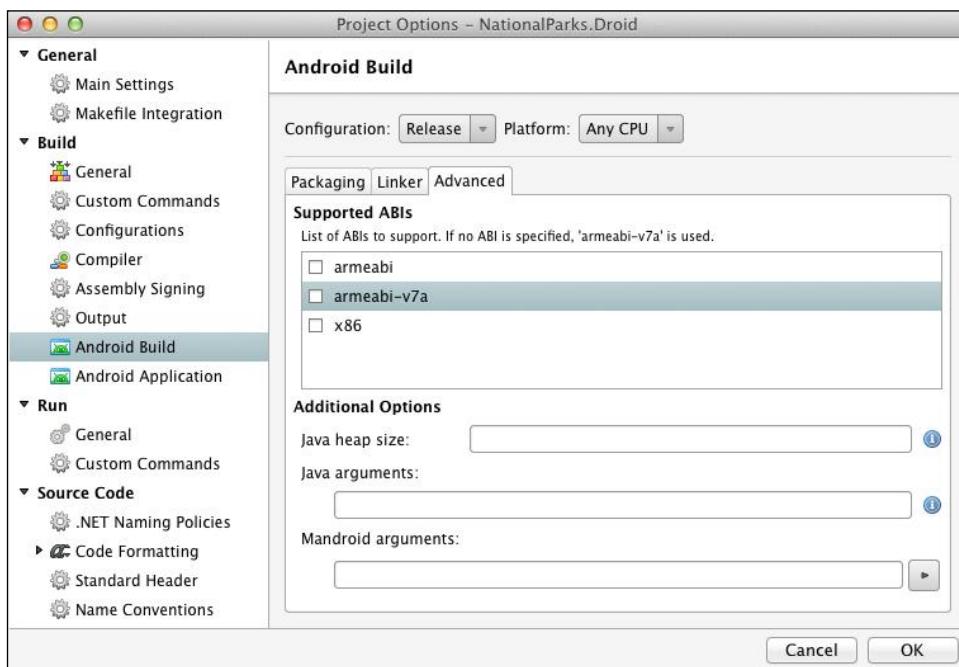
- Using a command-line option `linkskip`, for example,
`--linkskip=someassembly`

- Using the `AndroidLinkSkip` MSBuild property, as follows:

```
<PropertyGroup>
    <AndroidLinkSkip>Assembly1;Assembly2</AndroidLinkSkip>
</PropertyGroup>
```

Supported ABIs

Android supports a number of different CPU architectures. The Android platform defines a set of **Application Binary Interfaces (ABI)** that corresponds to different CPU architectures. By default, Xamarin.Android assumes that `armeabi-v7a` is appropriate for most circumstances. To support additional architectures, check each option that applies on the **Project Options** dialog box under the **Android Build** section.



Publishing a release APK

Now that you have adjusted the setting needed to produce a release build, you are ready to publish the actual APK. When we say publish, we simply mean to produce an APK that can be uploaded to the Google Play Store. The following sections discuss the steps of producing a signed APK from within Xamarin Studio.

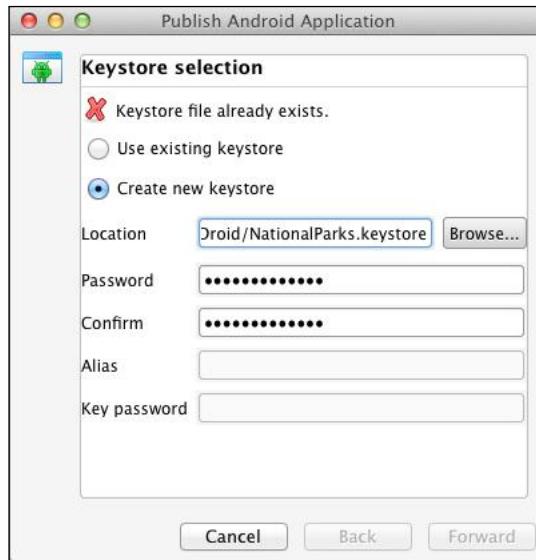
Keystores

A keystore is a database of security certificates created and managed by the keytool program from the Java SDK. You can create the keystore outside of Xamarin Studio using the keytool command or from within Xamarin Studio that provides a UI that interfaces with the keytool command. The next section takes you through the steps to publish an APK and create a new keystore from within Xamarin Studio.

Publishing from Xamarin.Android

The following steps guide you through the creation of a new keystore as a part of the process of creating a signed APK:

1. In the **Configuration** drop-down box, select **Release**.
2. Navigate to **Project | Publish Android Application** from the main menu; note the **Keystore selection** page of the **Publish Android Application** wizard, as shown in the following screenshot:

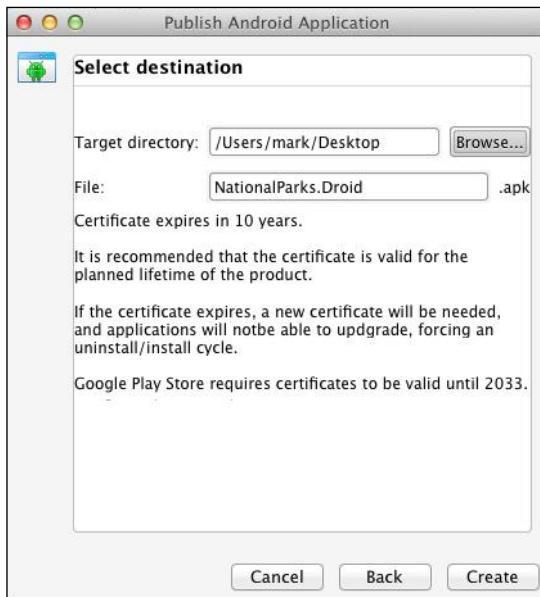


3. Select **Create new keystore**, select a location including a filename for the keystore, and enter the password and confirm it. The example keystore is in the project folder named `NationalParks.keystore` and the password is `nationalparks`.
4. Select **Forward**; you will see the **Key creation** page of the **Publish Android Application** wizard, as shown in the following screenshot:

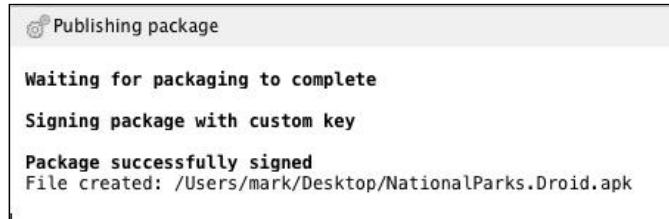


5. Enter all the relevant information. This example uses `nationalparks` for the **Alias** field and **Password**.

6. Select **Forward**; you will see the **Select destination** page of the **Publish Android Application** wizard, as shown in the following screenshot:



7. Select the required **Target directory** option and click on **Create**. Xamarin Studio will compile the app for release and generate a signed APK file. You should see the following in the **Publishing package** pad:



The resulting APK is ready for final testing and potential distribution. Be sure to secure and back up your keystore as it is critical to distributing future versions.

Republishing from Xamarin.Android

Subsequent publications of an app should always use the original keystore. To accomplish this, simply select **Use existing keystore** on the **Keystore selection** page of the **Publish Android Application** wizard.

Publishing from Visual Studio

Publishing a signed APK from within Visual Studio essentially follows the same process. To do so, simply navigate to **Tools | Publish Android Application** from the main menu.

App distribution options

Distributing Xamarin.Android apps is no different from any other Android app. They can be distributed through all the normal channels, app stores, e-mail attachments, website links, thumb drive, and so on.

Summary

In this chapter, we have reviewed activities related to preparing for distribution of an app and the process to actually produce a signed release APK file.

This chapter completes our book on Xamarin Essentials. We have tried to provide a productive approach for experienced mobile developers to quickly come up to speed on developing apps using the Xamarin platform. We have reviewed the Xamarin architecture, developed functional apps for both iOS and Android, and looked at how to maximize Xamarin's value by sharing code across mobile platforms using several different approaches and frameworks. You are now ready to put Xamarin to work.

I hope you have found this book a useful resource and that it has generated some excitement in you about developing great mobile apps with Xamarin.

Index

Symbols

.NET events, Objective-C delegates
mapping via 22, 23
.NET properties, Objective-C delegates
mapping via 23

A

AbsoluteLayout 159
action properties 31
Ad-Hoc and enterprise distribution 187-189
Ahead-of-Time compilation. *See*
 AOT compilation
Android Application
 (AndroidManifest.xml) settings 195
Android apps
 developing, Visual Studio used 15
 developing, Xamarin Studio used 13
Android apps, Xamarin.Forms 167
Android Device Monitor (ADM) 96
Android Emulator
 running, with NationalParks.Droid 85-87
Android implementation,
 DependencyService API
 creating 175, 176
AndroidManifest.xml file
 about 108
 settings, changing 194
Android user interface, MvvmCross
 Android app
 implementing 138, 139
Android Virtual Devices (AVD) 85
AOT compilation
 about 19

limitations 27, 28
URL 20

APK file
contents 109

AppDelegate.cs.txt class 149

app, DependencyService API
 running 177

Apple's Instruments app
 URL 182

Application Binary Interfaces (ABI) 199

ApplicationManifest.xml file
 about 41
 attributes 41, 42
 editor 42

App profiling 181

app start up, Xamarin.Forms framework
 about 166
 Android apps 167
 iOS apps 166
 shared app class 166

ARM thumb instruction set 187

assemblies
 skipping 198

AssemblyInfo.cs
 settings, changing 194

automatic collections 45

B

binding libraries
 creating 25

binding, modes
 OneTime binding 129
 OneWay binding 129
 OneWayToSource binding 129
 TwoWay binding 129

C

C#
about 11
URL 11

calls, DependencyService API
adding 177

CarouselPage 158

Cells
about 159
EntryCell 159
ImageCell 159
SwitchCell 159
TextCell 159

code
generating, for storyboard files 29
generating, for XIB 29
preserving, with attributes 197
preserving, with custom linker files 198

code-behind classes,
Xamarin.Forms 162, 163

Code Generation options
about 186
ARM thumb instruction set 187
LLVM optimizing compiler 187
supported architectures 186

code sharing techniques
cons 122
pros 122

collections
using, with *Xamarin.Android*
bindings 38, 39

commands 127, 128

Common Language Runtime (CLR) 7, 34

component store
URL 9

ContentPage 158

ContentView Layout 159

C# types
and type safety 21

D

Dalvik Virtual Machine (Dalvik VM) 34

data binding
about 128, 164
INotifyPropertyChanged interface 129

modes 129
specifications 130

data, sample national parks app
passing 69-71

debugging options 186

debug, Xamarin.Android apps
disabling 194

DeleteClicked action, sample national parks app
implementing 68

Delete() method 91

DependencyService API
about 165
Android implementation, creating 175, 176
app, running 177
calls, adding 177
interface, creating 174
iOS implementation, creating 174, 175
used, for showing directions 174
used, for showing photos 174

designer files 29

DetailActivity view,
NationalParks.Droid app
ActionBar items, adding 97
creating 96, 97
navigation, adding 99
populating 98
Show Directions action, handling 99
Show Photos action, handling 98

Detail.axml layout
updating 144

DetailView activity
creating 145

DetailViewController, sample national parks app
finishing 73, 74

DetailViewModel
creating 143

detail view, MvvmCross Android app
Detail.axml layout, updating 144
DetailView activity, creating 145
DetailViewModel, creating 143
implementing 143
navigation, adding 145

detail view, MvvmCross iOS app
implementing 151

developer environment
about 12
IDEs, comparing 16
version control 17, 18
Visual Studio environment, using 14
Visual Studio, used for developing
 Android apps 15
Visual Studio, used for developing
 iOS apps 15
Xamarin Studio environment, using 12
Xamarin Studio, used for developing
 Android apps 13
Xamarin Studio, used for developing
 iOS apps 14

Done.Clicked event handler, sample national parks app
implementing 67, 68

Dynamic-Link Library (DLL) 111

E

EditActivity, NationalParks.Droid app
ActionBar items, adding 101
creating 100
Delete action, handling 104
Edit action, navigating 105
ListView, refreshing in MainActivity 106
navigation, adding 105
New action, navigating 105
populating 102
reference variables, creating
 for widgets 102
Save action, handling 103

Edit.axml layout
updating 146

editor, ApplicationManifest.xml file 42

EditView activity
creating 147

EditViewController, sample national parks app
adding 65-67
finishing 74, 75

EditViewModel
creating 146

edit view, MvvmCross Android app
Edit.axml layout, updating 146
EditView activity, creating 146

EditViewModel, creating 146
implementing 146
navigation, adding 147, 148

edit view, MvvmCross iOS app
implementing 153
master view list, refreshing 154
navigation, adding 153, 154

entity class, sample national parks app
creating 57

enumerations, Xamarin.Android
 bindings 40

events
versus listeners 38

Extensible Application Markup Language. See **XAML**

F

file linking
about 112
shared files using, NationalParks.Droid
 updated for 115, 116
shared files using, NationalParks.iOS
 updated for 116, 117
shared library project, creating 112-114

FirstView class 149

Frame Layout 159

G

garbage collection (GC)
about 25, 43, 45
automatic collection 45
direct references, reducing in
 peer objects 45
JNI references 44
major collections 45
minor collections 45
Mono collections 44
peer objects, disposing 45

generated elements, NationalParks.Droid
 AndroidManifest.xml file 108
 APK file 108, 109
 peer objects 107
 reviewing 107

Genymotion
 NationalParks.Droid, running with 87
 URL 87

GetItemId() method 93

Git

- about 18
- for Visual Studio, URL 17

GitHub

- URL 126

global reference, JNI 44

Grid Layout 159

I

IDEs

- comparing 16

IFileHandler

- implementing 119

inheritance

- usage 21

INotifyPropertyChanged interface

- about 129
- implementing 137, 138

interface, DependencyService API

- creating 174

interfaces, Xamarin.Android bindings 39

iOS apps

- about 166
- developing, Visual Studio used 15, 16
- developing, Xamarin Studio used 14
- settings 181

iOS App Store Package. *See IPA*

iOS Build settings

- about 181
- Code Generation options 186
- debugging options 186
- Linker Options 184
- SDK Options 183

iOS implementation,

DependencyService API

- creating 174, 175

iOS user interface, MvvmCross iOS app

- implementing 150

IPA

- creating 187

J

Java Native Interface (JNI) 34

Java objects 43

JNI references

- global reference 44

- weak reference 44

JSON-formatted file, sample national

parks app

- adding 58

- objects, loading from 59

- objects, saving to 60

Json.NET, sample national parks app

- adding 57

K

keystores 200

L

layouts

- about 159
- AbsoluteLayout 159
- ContentView layout 159
- Frame 159
- Grid 159
- RelativeLayout 159
- ScrollView 159
- StackLayout 159

linker options

- about 184, 185, 196
- Don't link option 184, 196
- Link all assemblies option 185, 197
- linker, overriding 185, 186, 197
- Link SDK assemblies only option 185, 196

linker, overriding

- assemblies, skipping 186, 198
- code, preserving 185
- code, preserving with attributes 197
- code, preserving with custom linker files 198

- Link Away, disabling 186

listeners

- versus events 38

LLVM optimizing compiler 187

M

MainActivity.cs file,

NationalParks.Droid app 82, 83

MainActivity, NationalParks.Droid app
action, adding to ActionBar 94, 95
adapter, creating 92-94
enhancing 91
ListView instance, adding 91
Main.xml file, editing 92
OnCreateOptionsMenu() method,
 overriding 95
OnOptionsItemSelected() method,
 overriding 95
Xamarin.Android Designer, touring 91

Main.axml file, NationalParks.Droid app 83

major collections 43, 45

Managed Callable Wrappers (MCW) 35

managed objects 43

Master.axml layout
 updating 141, 142

MasterDetailPage 158

master list view, MvvmCross Android app
 implementing 139
 Master.axml layout, updating 141, 142
 MasterView activity, creating 142, 143
 MasterViewModel, creating 139, 140

MasterView activity
 creating 142, 143

MasterViewModel
 creating 139, 140

master view, MvvmCross iOS app
 implementing 150, 151

memory management 25-27

minor collections 43, 45

Model-View-ViewModel (MVVM) 126

Mono assemblies
 about 7, 20, 36
 reference link 36

Mono CLR
 about 7, 34
 peer objects 35

Mono collections 44

Mono's simple generational garbage collector
 major collections 43
 minor collections 43
 URL 43

MonoTouch.Dialog (MT.D)
 about 76
 URL 76

MonoTouch Profiler
 URL 182

MvvmCross Android app
 Android user interface,
 implementing 138, 139
 creating 135, 136
 detail view, implementing 143
 edit view, implementing 146
 INotifyPropertyChanged interface,
 implementing 137, 138
 master list view, implementing 139
 NationalParks.IO, reusing 137
 NationalParks.PortableData, reusing 137

MvvmCross core project
 creating 134, 135

MvvmCross iOS app
 cons 155
 creating 149
 detail view, implementing 151
 edit view, implementing 153
 iOS user interface, implementing 150
 master view, implementing 150, 151
 pros 155

MVVM pattern 126

N

NationalParksData singleton
 creating 89, 90

NationalParks.Droid app
 creating 80, 81
 debugging, with Xamarin Studio 84
 running 96, 100, 106
 running, on physical device 87
 running, with Android Emulator 85-87
 running, with Genymotion 87
 running, with Xamarin Studio 84
 updated, for using PCL 120, 121
 updated, for using shared files 115, 116

NationalParks.Droid app, extending
 about 88
 entity class, borrowing 89
 JSON file, borrowing 89
 Json.NET, adding 88
 NationalParksData singleton,
 creating 89, 90
 national parks, storing 88

NationalParks.Droid app, reviewing
about 82
MainActivity.cs file 82, 83
Main.axml file 83
project folders 84
Resource.designer.cs file 82
Resources folder 82
Xamarin Studio preferences 84

NationalParks.Droid, extending
about 88
DetailActivity view, creating 96, 97
EditActivity, creating 100
MainActivity, enhancing 91
national parks, loading 88
national parks, storing 88

NationalParks.IO
reusing 137

NationalParks.iOS
updated, for using PCL 121, 122
updated, for using shared files 116, 117

NationalParks.MvvmCross
MvvmCross Android app, creating 135, 136
MvvmCross core project, creating 134, 135
MvvmCross iOS app, creating 149

NationalParks.PortableData
adding 169
creating 118, 119
reusing 137

NationalParks Xamarin.Forms app
creating 168
NationalParks.PortableData, adding 169
ParkDetailPage, implementing 172, 173
ParkEditPage, implementing 177-179
ParksListPage, implementing 170-172
solution, creating 168, 169

navigation
about 160
PopAsync() method 160
PopModalAsync() method 160
PopToRootAsync() method 160
PushAsync() method 160
PushModalAsync() method 160

navigation, MvvmCross Android app
adding 147, 148

navigation, MvvmCross iOS app
adding 152

NavigationPage 158

nested classes
mapping 40

non-designer files 30

O

Objective-C delegates
about 22
via .NET events 22, 23
via .NET properties 23
via strongly typed delegates 23, 24
via weakly typed delegates 24, 25

objects
disposing 27
Java objects 43
managed objects 43
peer objects 43
preventing, from destroyed 27

OnCreateOptionsMenu() method 95

OneTime binding 129

OneWay binding 129

OneWayToSource binding 129

OnOptionsItemSelected() method 95

OS X
Xamarin, installing 12

P

packages.config file 149

Pages
about 158
CarouselPage 158
ContentPage 158
MasterDetailPage 158
NavigationPage 158
TabbedPage 158

ParkDetailPage
implementing 172, 173

ParkEditPage
implementing 177-179

ParksListPage
implementing 170-172

peer objects
about 35, 43
direct references, reducing 45
disposing 45

PopAsync() method 160

PopModalAsync() method 160
PopToRootAsync() method 160
Portable Class Libraries (PCL)
 about 118
 IFileHandler, implementing 119
 NationalParks.PortableData, creating 118
 using, NationalParks.Droid updated
 for 120, 121
 using, NationalParks.iOS updated
 for 121, 122
product information
 URL 9
product suite, Xamarin
 about 8
 business 8
 component store 9
 enterprise 8
 indie 8
 information 9
 offerings 8
 pricing 9
 starter 8
 Xamarin Test Cloud 9
project options,
 NationalParks.Droid app 84
Project Options view, sample
 national parks app
 about 51
 iOS Application 51
 iOS Build 51
 iOS Bundle Signing 51
 iOS IPA Options 51
project, Xamarin.Forms framework
 organizing 167, 168
PushAsync() method 160
PushModalAsync() method 160

R

RelativeLayout 159
release APK, preparing for
 Android Application
 (AndroidManifest.xml) settings 195
 AndroidManifest.xml settings,
 changing 194
 Application Binary Interfaces (ABI) 199
 AssemblyInfo.cs settings, changing 194

debug, disabling 194
Linker Options settings 196
Xamarin.Android apps, profiling 194

release APK, publishing
 about 200
 keystores 200
 Visual Studio, publishing from 203
 Xamarin.Android, publishing
 from 200-202
 Xamarin.Android, republishing from 203

renderers
 using 165
Resource.designer.cs file,
 NationalParks.Droid app 82

Resources folder,
 NationalParks.Droid app 82

resources, Xamarin.Android bindings 41

Runnable interface
 mapping 40

S

sample national parks app
 about 48
 creating 48-51
 data, passing 69-71
 DeleteClicked action, implementing 68, 69
 detail view 48
 DetailViewController, finishing 73, 74
 Done.Clicked event handler,
 implementing 67, 68
 edit view 48
 EditViewController, adding 65-67
 EditViewController, finishing 74, 75
 entity class, creating 57
 extending 56
 finishing 72
 JSON-formatted file, adding 58
 Json.NET, adding 57
 list view 48
 loading 56
 MonoTouch.Dialog (MT.D) 76
 objects, loading from JSON-formatted
 file 59
 objects, saving to JSON-formatted file 60
 Project Options view 51
 running 60, 72, 76

segues, adding 65-67
storing 56
UI, enhancing 60, 61
Xamarin.iOS Designer, touring 62-65
Xamarin Studio, debugging within 52-56
Xamarin Studio, running within 52-56
Save() method 90
ScrollView Layout 159
SDK Options 183
segues, sample national parks app
 adding 65-67
Setup class 149
shared files
 using, NationalParks.Droid updated
 for 115, 116
 using, NationalParks.iOS updated
 for 116, 117
shared library project
 creating 112-114
sharing game 111, 112
solution, NationalParks Xamarin.Forms app
 creating 168, 169
StackLayout 159
startup process, MvvmCross apps
 about 132
 Android startup 133
 App.cs 132
 iOS startup 133
 Setup.cs 133
storyboard files. *See XIB*
strongly typed delegates, Objective-C delegates
 mapping via 23, 24
Subversion 18

T

TabPage 158
Team Foundation Server (TFS)
 about 17, 18
 add-in for Xamarin Studio, URL 17
TestFlight distribution
 about 189, 190
 URL 190
TwoWay binding 129

U

UI, sample national parks app
 app, running 72
 data, passing 69-71
 DeleteClicked action, implementing 67, 68
 Done.Clicked event handler,
 implementing 67, 68
 EditViewController, adding 65-67
 enhancing 60, 61
 segues, adding 65-67
 Xamarin.iOS Designer, touring 62-65

V

version control 17
View 126, 127, 158
ViewModels
 about 126, 127
 navigating between 131
 parameters, passing 131
 solution/project organization 132
 startup process 132
Visual Studio
 cons 16
 pros 16
 publishing from 203
 used, for developing Android apps 15
 used, for developing iOS apps 15, 16
 Xamarin.Android projects, working
 with 107
Visual Studio environment 14

W

weakly typed delegates, Objective-C delegates
 mapping via 24, 25
weak reference, JNI 44
Windows
 Xamarin, installing 12
Windows Presentation Foundation (WPF) 127, 161, 164

X

Xamarin

benefits 9, 10
drawbacks 10, 11
installing 11
installing, on OS X 12
installing, on Windows 12
product suite 8
URL 8

Xamarin.Android

folders 83
publishing from 200-202
republishing from 203

Xamarin.Android app

distribution, options 203
packaging 36
profiling 194
sample app, creating 80

Xamarin.Android bindings

about 36
collections 38, 39
design principles 37
enumerations 40
events, versus listeners 38
interfaces 39
nested classes, mapping 40
properties 37
resources 41
Runnable interface, mapping 40

Xamarin.Android Designer 46

Xamarin.Android projects

working, in Visual Studio 107

Xamarin.Forms framework

about 157, 158
app startup 166
cells 159
code-behind classes 162, 163
cons 179
data binding 164
DependencyService API 165
Layouts 159
native features 165
Navigation 160
Pages 158

project, organizing 167, 168
pros 179

renderers, using 165
Views 158

Xamarin.Forms, user interfaces

about 160
declarative approach 160
programmatic approach 160

Xamarin.iOS

about 19
runtime features, disabled 28
URL 20

Xamarin.iOS apps

iOS Application (Info.plist) settings 182
profiling 182

publishing, to App store 191

Xamarin.iOS apps, distributing

about 181, 187
Ad-Hoc and enterprise
distribution 187-189
App profiling 181
App Store submission 191
Code Generation options 186
debugging options 186
iOS Application settings 181
iOS Build settings 181, 183
Linker Options 184
SDK Options 183
TestFlight distribution 189, 190

Xamarin.iOS bindings

about 20
C# types and type safety 21
design principles 20, 21
inheritance, using 21

Xamarin.iOS Designer

about 31, 32
features 32
touring 62-65
URL 62

Xamarin Studio

cons 16
debugging, within 52-56
environment, using 12, 13
preferences 84
pros 16

running, within 53-56
used, for debugging Xamarin.Android
 apps 84
used, for developing Android apps 13
used, for developing iOS apps 14
used, for running Xamarin.Android
 apps 84

Xamarin Test Cloud

 URL 9

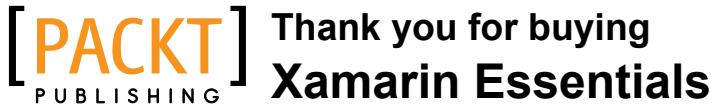
XAML 161, 162

Xcode Interface Builder

 Apple tutorial, URL 62
 Xamarin tutorial, URL 62

XIB

 action properties 31
 code, generating for 29
 designer files 29
 generated classes 29
 non-designer files 30
 outlets properties 31



**Thank you for buying
Xamarin Essentials**

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at 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, then please 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.

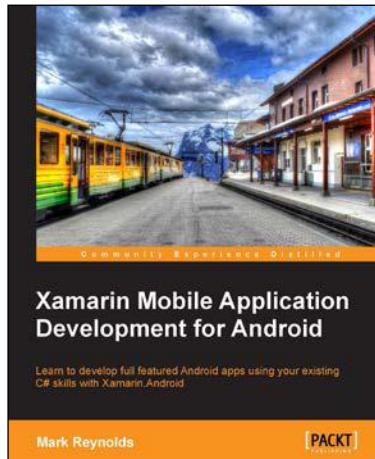


Xamarin Cross-platform Application Development

ISBN: 978-1-84969-846-7 Paperback: 262 pages

Develop production-ready applications for iOS and Android using Xamarin

1. Write native iOS and Android applications with Xamarin.
2. Add native functionality to your apps such as push notifications, camera, and GPS location.
3. Learn various strategies for cross-platform development.



Xamarin Mobile Application Development for Android

ISBN: 978-1-78355-916-9 Paperback: 168 pages

Learn to develop full featured Android apps using your existing C# skills with Xamarin.Android

1. Gain an understanding of both the Android and Xamarin platforms.
2. Build a working multi-view Android app incrementally throughout the book.
3. Work with device capabilities such as location sensors and the camera.

Please check www.PacktPub.com for information on our titles



Xamarin Mobile Application Development for iOS

ISBN: 978-1-78355-918-3 Paperback: 222 pages

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

1. A clear and concise look at how to create your own apps building on what you already know of C#.
2. Create advanced and elegant apps by yourself.
3. Ensure that the majority of your code can also be used with Android and Windows Mobile 8 devices.



iOS 7 Game Development

ISBN: 978-1-78355-157-6 Paperback: 120 pages

Develop powerful, engaging games with ready-to-use utilities from Sprite Kit

1. Pen your own endless runner game using Apple's new Sprite Kit framework.
2. Enhance your user experience with easy-to-use animations and particle effects using Xcode 5.
3. Utilize particle systems and create custom particle effects.

Please check www.PacktPub.com for information on our titles