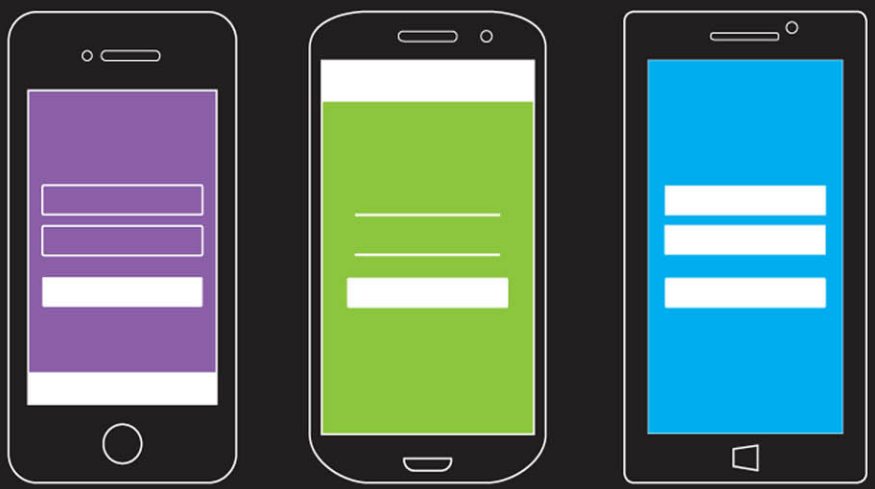
Xamarin

Microsoft

Creating Mobile Apps with Xamarin.Forms



PREVIEW EDITION

Cross-platform C# programming for iOS, Android, and Windows Phone

CHARLES PETZOLD

PREVIEW EDITION

This excerpt provides early content from a book currently in development and is still in draft format. See additional notice below.

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN: 978-0-7356-9725-6

Microsoft Press books are available through booksellers and distributors worldwide. Please tell us what you think of this book at http://aka.ms/tellpress.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Project Editor: Devon Musgrave
Cover illustration: Serena Zhang

Table of contents

Introduction

This is a Preview Edition of a book about writing applications for Xamarin.Forms, the exciting new mobile development platform for iOS, Android, and Windows Phone unveiled by Xamarin in May 2014. Xamarin.Forms lets you write shared user-interface code in C# and XAML (the eXtensible Application Markup Language) that maps to native controls on these three platforms.

This book is a Preview Edition because it's not complete. It has only six chapters. We anticipate that the final version of the book will have at least half a dozen additional chapters and that the chapters in this Preview Edition might be fleshed out, enhanced, or completely reconceived. The final edition of the book will probably be published in the spring of 2015.

Who should read this book

This book is for C# programmers who want to write applications for the three most popular mobile platforms: iOS, Android, and Windows Phone with a single code base. Xamarin.Forms also has applicability for those programmers who want eventually to use C# and the Xamarin.iOS and Xamarin.Android libraries to target the native application programming interfaces (APIs) of these platforms. Xamarin.Forms can be a big help in getting started with these platforms or in constructing a prototype or proof-of-concept application.

Assumptions

This book assumes that you know C# and have some familiarity with the use of the .NET Framework. However, when I discuss some C# and .NET features that might be somewhat new to recent C# programmers, I adopt a somewhat slower pace. In particular, the introduction of the `async` keyword and `await` operator in Chapter 3 follows a discussion that shows how to do asynchronous programming using traditional callback methods.

Organization of this book

This book is intended as a tutorial to learn Xamarin.Forms programming. It is not a replacement for the online API documentation, which can be found here under the heading Xamarin.Forms Framework on this page: http://api.xamarin.com/.

This Preview Edition's Chapter 1 discusses Xamarin.Forms in the larger context of mobile development and the Xamarin platform and also covers the hardware and software configurations you'll need. Chapter 2 explores some of the basics of Xamarin.Forms programming, including the use of `Label`, `Button`, and `StackLayout`.

In Chapters 3 and 4, however, I tried to do something a little different: These chapters show the progressive step-by-step development of a small Xamarin.Forms application. Despite the simplicity of this program, it is in many ways a "real" application, and requires essential real-app facilities such as file I/O and application lifecycle handling, both of which turned out to be somewhat more challenging than I originally anticipated. I'm curious to hear whether these two chapters "work" or not. See the section below on submitting feedback to us.

Chapters 5 and 6 return to more conventional API tutorials. My biggest regret is that I wasn't able to get some coverage of XAML into this Preview Edition. However, the Xamarin website has some additional resources for learning Xamarin.Forms including a six-part "XAML for Xamarin.Forms" document: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/.

Conventions and features in this book

This book has just a few typographical conventions:

- All programming elements referenced in the text—including classes, methods, properties, variable names, etc.—are shown in a monospaced font, such as the `StackLayout` class.
- Items that appear in the user interface of Visual Studio or Xamarin Studio, or the applications discussed in these chapters, appear in boldface, such as the **Add New Project** dialog.
- Application solutions and projects also appear in boldface, such as **ColorScroll**.

System requirements

This book assumes that you'll be using Xamarin.Forms to write applications that simultaneously target all three supported mobile platforms—iOS, Android, and Windows Phone. However, it's very likely that many readers will be targeting only one or two platforms in their Xamarin.Forms solutions. The platforms you target—and the Xamarin Platform package you purchase—govern your hardware and software requirements. For targeting iOS devices, you'll need a Mac installed with Apple XCode as well as the Xamarin Platform, which includes Xamarin Studio. For targeting Windows Phone, you'll need Visual Studio 2012 or 2013 (not an Express Edition) on a PC, and you'll need to have installed the Xamarin Platform.

However, you can also use Visual Studio on the PC to target iOS devices if the Mac with XCode and the Xamarin Platform is accessible via WiFi. You can target Android devices from Visual Studio on the PC or from Xamarin Studio on either the PC or Mac.

Chapter 1 has more details on the various configurations you can use, and resources for additional information and support. My setup for creating this book consisted of a Microsoft Surface Pro 2 (with external monitor, keyboard and mouse) installed with Visual Studio 2013 and the Xamarin Platform,

connected by WiFi with a MacBook Pro installed with XCode and the Xamarin Platform.

Downloads: Code samples

The sample programs shown in the pages of this book were compiled in early September with version 1.2.2.6243 of Xamarin.Forms. The source code of these samples is hosted on a repository on GitHub: https://github.com/xamarin/xamarin-forms-book-preview/.

You can clone the directory structure to a local drive on your machine, or download a big ZIP file. I'll try to keep the code updated with the latest release of Xamarin.Forms and to fix (and comment) any errors that might have sneaked through.

You can report problems, bugs, or other kinds of feedback about the book or source code by clicking the **Issues** button on this GitHub page. You can search through existing issues or file a new one. To file a new issue, you'll need to join GitHub (if you haven't already).

Use this GibHub page only for issues involving the book. For questions or discussions about Xamarin.Forms itself, use the Xamarin.Forms forum: http://forums.xamarin.com/categories/xamarin-forms.

Updating the code samples

The libraries that comprise Xamarin.Forms are distributed via the NuGet package manager. The Xamarin.Forms package consists of five dynamic-link libraries: Xamarin.Forms.Core.dll, Xamarin.Forms.Xaml.dll, Xamarin.Forms.Platform.iOS.dll, Xamarin.Forms.Platform.Android.dll, and Xamarin.Forms.Platform.WP8.dll. The Xamarin.Forms package also requires Xamarin Support Library v4 (Xamarin.Android.Support.v4.dll) and the Windows Phone Toolkit (Microsoft.Phone.Controls.-Toolkit.dll), which should be automatically included.

When you create a new Xamarin.Forms solution using Visual Studio or Xamarin Studio, a version of the Xamarin.Forms package becomes part of that solution. However, that might not be the latest Xamarin.Forms package available from NuGet. Here's how to update that package:

In Visual Studio, right-click the solution name in the **Solution Explorer** and select **Manage NuGet Packages for Solution**. Select **Installed packages** at the left of the dialog to see what's currently installed, and **Updates** and **nuget.org** at the left to choose to update the package. If an update is available, clicking the **Update All** button is easiest to get it into the solution.

In Xamarin Studio, in the individual projects in the **Solution** list, under **Packages**, select the Xamarin.Forms package and select **Update** from the tool dropdown.

The source code for this book that is stored on GitHub does not include the actual NuGet packages. Xamarin Studio will automatically download them when you load the solution, but Visual Studio will not. After you first load the solution into Visual Studio, right-click the solution name in the **Solution**

Explorer and select **Manage NuGet Packages for Solution**. You should be given the option to restore the packages with a **Restore** button at the top of the dialog. You can then update the package by selecting **Updates** and **nuget.org** at the left and (if an update exists) pressing the **Update All** button.

Acknowledgments

It's always seemed peculiar to me that authors of programming books (such as this one) are sometimes better known to programmers than the people who actually created the product that is the subject of the book! The real brains behind Xamarin.Forms are Jason Smith, Eric Maupin, Stephane Delcroix, and Seth Rosetter. Congratulations, guys! We've been enjoying the fruits of your labor!

Over the months that this Preview Edition was in progress, I have benefited from valuable feedback, corrections, and edits from several people. This book wouldn't exist without the collaboration of Bryan Costanich at Xamarin and Devon Musgrave at Microsoft Press. Both Bryan and Craig Dunn at Xamarin read some of my drafts of early chapters and easily persuaded me to take a somewhat different approach to the material. Later on, Craig kept me on track. During the final days before my deadline for this Preview Edition, Stephane Delcroix offered essential technical reads and John Meade performed copyediting under extreme crunch conditions. Microsoft Press supplemented that with another very helpful technical read by Andy Wigley, who persistently prodded me to make the book better.

Almost nothing I do these days would be possible with the daily companionship and support of my wife, Deirdre Sinnott.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at http://aka.ms/mspressfree.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at http://aka.ms/tellpress. Your feedback goes directly to the editors at Microsoft Press. (No personal information will be requested.) Thanks in advance for your input!

CHAPTER 1

How does Xamarin.Forms fit in?

There is much joy in programming. There is joy in analyzing a problem, breaking it down into pieces, formulating a solution, mapping out a strategy, approaching it from different directions, and crafting the code. There is very much joy in seeing the program run for the first time, and then more joy in eagerly diving back into the code to make it better and faster.

There is also often joy in hunting down bugs, in ensuring that the program runs smoothly and predictably. Few things are quite as joyful as finally identifying a particularly recalcitrant bug and definitively stamping it out.

There is even joy in realizing that the original approach you took is not quite the best. Many developers discover that they've learned a lot writing a program, including that there's a better way to structure the code. Sometimes, a partial or even total rewrite can result in a much better application. The process is like standing on one's own shoulders, and there is much joy in attaining that perspective and knowledge.

However, not all aspects of programming are quite so joyful. One of the nastier programming jobs is taking an existing working program and rewriting it in an entirely different programming language, or porting it to another operating system with an entirely different application programming interface (API).

A job like that can be a real grind. Yet, such a rewrite may very well be necessary: The application that's been so popular on the iPhone might be even more popular on Android devices, and there's only one way to find out.

But here's the problem: As you're going through the original source code and moving it to the new platform, do you maintain the same program structure as you go along so that the two versions exist in parallel? Or do you try to make improvements and enhancements?

The temptation, of course, is to entirely rethink the application and make the new version better. But the further the two versions drift apart, the harder they will be to maintain in the future.

For this reason, an inherent dread pervades the forking of one application into two. With each line of code you write, you realize that all the future maintenance work, all the future revisions and enhancements, have become two jobs rather than one.

This is not a new problem. For over half a century, developers have craved the ability to write a single program that runs on multiple machines. This is one of the reasons that high-level languages were invented in the first place, and this is why the concept of "cross-platform development" continues to exert a powerful attraction for programmers.

Cross-platform mobile development

The personal computer industry has experienced a massive shift in recent years. Desktop computers still exist, of course, and they remain vital for tasks that require keyboards and large screens: programming, writing, spread-sheeting, data tracking. But much of personal computing now occurs on smaller devices, particularly for information lookup and media consumption applications. Tablets and smartphones have a fundamentally different user-interaction paradigm based primarily on touch with a keyboard that pops up only when necessary.

The mobile landscape

Although the mobile market has the potential of rapid change, there are currently two major phone and tablet platforms:

- The Apple family of iPhones and iPads, all of which run the iOS operating system.
- The Android operating system developed by Google based on the Linux kernel, which runs on a variety of phones and tablets.

How the world is divided between these two giants depends on how they are measured. There are more Android devices out in the wild, but the iPhone and iPad users are more devoted and spend more time with their devices.

There is also a third mobile development platform that is not as popular as iOS and Android but that involves a company with a strong history in the personal computer industry:

- Microsoft's Windows Phone.

Windows Phone recently got a boost with the recent releases of Windows 8.1 and Windows Phone 8.1. With these releases, a single application programming interface called the Windows Runtime is now available for applications running on desktop machines, laptops, tablets, and phones.

For software developers, the optimum strategy is to target more than just one of these platforms. But that's not easy. There are four big obstacles:

Problem 1: Different user-interface paradigms

All three platforms incorporate similar ways of presenting the graphical user interface (GUI) and interaction with the device through multi-touch, but in detail there are many differences. Each platform has different ways to navigate around applications and pages, different conventions for the presentation of data, different ways to invoke and display menus, and even different approaches to touch.

Users become accustomed to interacting with applications on a particular platform and expect to leverage that knowledge with future applications as well. Each platform acquires its own "culture" of sorts, and these cultural conventions then influence developers.

Problem 2: Different development environments

Programmers today are accustomed to working in a sophisticated integrated development environment (IDE). Such IDEs exist for all three platforms, but of course they are different:

- For iOS development, XCode on the Mac.
- For Android development, Eclipse on a variety of platforms.
- For Windows Phone development, Visual Studio on the PC.

Problem 3: Different programming interfaces

All three of these platforms are based on different operating systems with different APIs. In many cases, the three platforms all implement similar types of user-interface objects but with different names.

For example, all three platforms have something that lets the user toggle between two states:

- On the iPhone, it's a "view" called `UISwitch`.
- On Android devices, it's a "widget" called `Switch`.
- On Windows Phone, one possibility is a "control" called `ToggleSwitchButton` from the Windows Phone Toolkit NuGet package.

Of course, the differences go far beyond the names into the programming interfaces.

Problem 4: Different programming languages

Developers have some flexibility in choosing a programming language for each of these three platforms, but in general each platform is very closely associated with a particular programming language:

- Objective-C for the iPhone
- Java for Android devices
- C# for Windows Phone

These three languages are cousins of sorts because they are all object-oriented descendants of C, but they have become rather distant cousins.

For these reasons, a company that wishes to target multiple platforms might very well employ three different programmer teams, each team skilled and specialized in a particular language and API.

This language problem is particularly nasty, but it's the problem that is the most tempting to solve: If you could use the same programming language for these three platforms, you could at least share some code between the platforms. This shared code likely wouldn't be involved with the user-interface because each platform has different APIs, but there might well be application code that doesn't touch

the user interface at all.

A single language for these three platforms would certainly be convenient. But what language would that be?

The C# and .NET solution

A roomful of programmers would come up with a variety of answers to the question just posed, but a good argument can be made in favor of C#. C#, unveiled by Microsoft in the year 2000, is a fairly new programming language, at least when compared with Objective-C and Java. At first, C# seemed to be a rather straightforward strongly-typed imperative object-oriented language, certainly influenced by C++ (and Java as well) but with a much cleaner syntax than C++ and none of the historical baggage. In addition, the first version of C# had language-level support for properties and events, which turn out to be member types that are particularly suited for programming graphical user interfaces.

But C# has continued to grow and get better over the years. The support of generics, lambda functions, LINQ, and asynchronous operations has successfully elevated C# to be classified as a "multi-paradigm programming language" (at least by Wikipedia). C# code can be traditionally imperative, or flavored with declarative or functional programming paradigms.

On April 3, 2014, C# creator Anders Hejlsberg stood on a stage at the Microsoft Build 2014 conference and clicked a button that published an open-source version of the C# compiler called the .NET Compiler Platform (formerly known by its code name "Roslyn").

Since its inception, C# has been closely associated with the Microsoft .NET Framework. At the lowest level, .NET provides an infrastructure for the C# basic data types (`int`, `double`, `string`, and so forth). But .NET also provides an extensive .NET Framework class library for many common chores encountered in many different types of programming. These include:

- Math
- Debugging
- Reflection
- Collections
- Globalization
- File I/O
- Networking
- Security
- Threading

- Web services
- Data handling
- XML reading and writing

Here's another big reason for C# and .NET to be regarded as a compelling cross-platform solution:

It's not just hypothetical. It's a reality.

Shortly after the Microsoft's announcement of .NET in June 2000, the company Ximian (founded by Miguel de Icaza and Nat Friedman) initiated an open-source project called Mono to create an alternative implementation of the C# compiler and the .NET Framework that could run on Linux.

A decade later in 2011, the founders of Ximian (which had been acquired by Novell) founded Xamarin, which still contributes to the open-source version of Mono but has also adapted Mono to form the basis of cross-platform mobile solutions.

Shortly after the open-source version of C# was published, Scott Guthrie of Microsoft announced the formation of the .NET Foundation, which serves as a steward for open-source .NET technologies, in which Xamarin plays a major part.

A single language for all platforms

For the first three years of its existence, Xamarin mostly focused on three basic sets of .NET libraries:

- Xamarin.Mac, also known as MonoMac.
- Xamarin.iOS, also known as MonoTouch.
- Xamarin.Android, also known as Mono for Android or (more informally) MonoDroid.

Collectively, these libraries are known as the Xamarin Platform. The libraries consist of .NET versions of the native Mac, IOS, and Android APIs. Programmers using these libraries can write applications in C# to target the native APIs of these three platforms, but also (as a bonus) with access to the .NET Framework class library.

Xamarin also makes available Xamarin Studio, an integrated development environment that runs on both the Mac and PC and that lets you develop iPhone and Android applications on the Mac, and Android applications on the PC.

It is also possible to use Visual Studio with the Xamarin libraries to develop Mac, iPhone, and Android applications. (However, Mac and iPhone development also requires a Mac with XCode and Xamarin Studio installed and connected by WiFi with the PC.)

Of course, if you're using Visual Studio, you can also target Windows Phone and other Microsoft platforms.

Sharing code

The advantage of targeting multiple platforms with a single programming language comes from the ability to share code among the applications.

Before code can be shared, it must be structured for that purpose. Particularly since the widespread use of graphical user interfaces, programmers have understood the importance of separating application code into functional layers. Perhaps the most useful division is between user interface code and the underlying data models and algorithms. The popular MVC (Model-View-Controller) application architecture formalizes this code separation into a Model (the underlying data), the View (the visual representation of the data), and the Controller (which handles input from the user).

MVC originated in the 1980s. More recently, the MVVM (Model-View-ViewModel) architecture has effectively modernized MVC based on modern GUIs. MVVM separates code into the Model (the underlying data), the View (the user interface, including visuals and input), and the ViewModel (which manages data passing between the Model and the View).

When developing a program that targets multiple mobile platforms, the MVVM architecture helps guide the developer into separating code into the platform-specific View—the code that requires interacting with the platform APIs—and the platform-independent Model and ViewModel.

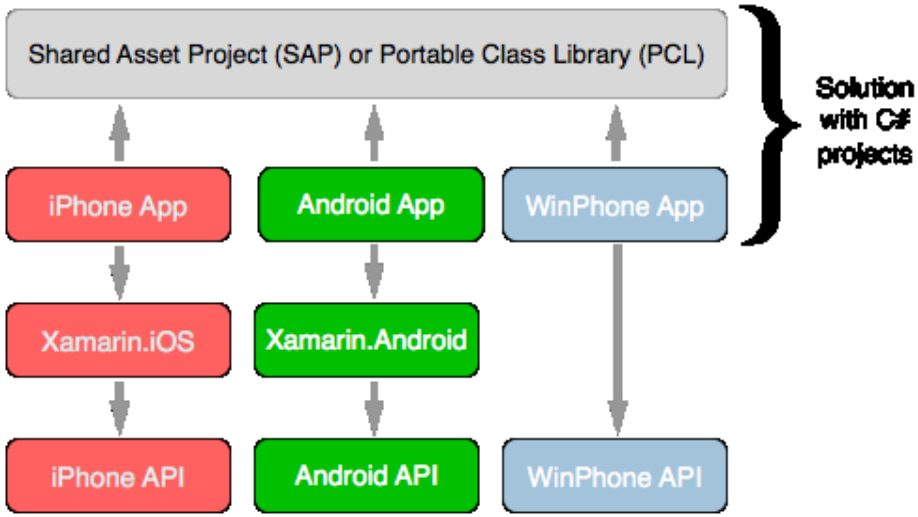
Often this platform-independent code needs to access files or the network, or use collections or threading. Normally these jobs would be considered part of an operating system API, but they are also jobs that can make use of the .NET Framework class library, and if the class library is available on each platform, it is effectively platform-independent.

The part of the application that is platform-independent can then be isolated, and—in the context of Visual Studio or Xamarin Studio—put into a separate project. This can be either a Shared Asset Project (SAP)—which simply consists of code and other asset files accessible from other projects—or a Portable Class Library (PCL), which encloses all the common code in a dynamic-link library (DLL) that can then be referenced from other projects.

Whichever method is used, this common code has access to the .NET Framework class library, so it can perform file I/O, handle globalization, access web services, decompose XML, and so forth.

This means that you can create a single Visual Studio solution that contains four C# projects to target the three major mobile platforms, or you can use Xamarin Studio to target iPhone and Android devices.

Here's a diagram roughly showing the interrelationships between the Visual Studio or Xamarin Studio projects, the Xamarin libraries, and the platform APIs:



The boxes in the second row are the actual platform-specific applications. These make calls into the common project and also (in the case with the iPhone and Android), the Xamarin libraries that implement the native platform APIs.

But the diagram is not quite complete: It doesn't show the SAP or PCL making calls to the .NET Framework class library. The PCL has access to its own version of .NET, while the SAP uses the version of .NET incorporated into each particular platform.

In this diagram, the Xamarin.iOS and Xamarin.Android libraries seem to be substantial, and while they are certainly important, they're mostly just language bindings and do not significantly add any overhead to API calls.

When the iPhone app is built, the Xamarin C# compiler generates C# Intermediate Language (IL) as usual but then makes use of the Apple compiler on the Mac to generate native iPhone machine code just like the Objective-C compiler. The calls from the app to the iPhone APIs are the same as if the application were written in Objective-C.

For the Android app, the Xamarin C# compiler generates IL, which runs on a version of Mono on the device alongside the Java engine, but the API calls from the app are pretty much the same as if the app were written in Java.

For mobile applications that have very platform-specific needs but also a potentially shareable chunk of platform-independent code, Xamarin.iOS and Xamarin.Android provide excellent solutions. You have access to the entire platform API, with all the power (and responsibility) that implies.

But for applications that might not need quite so much platform specificity, there is now an alternative that will simplify your life even more.

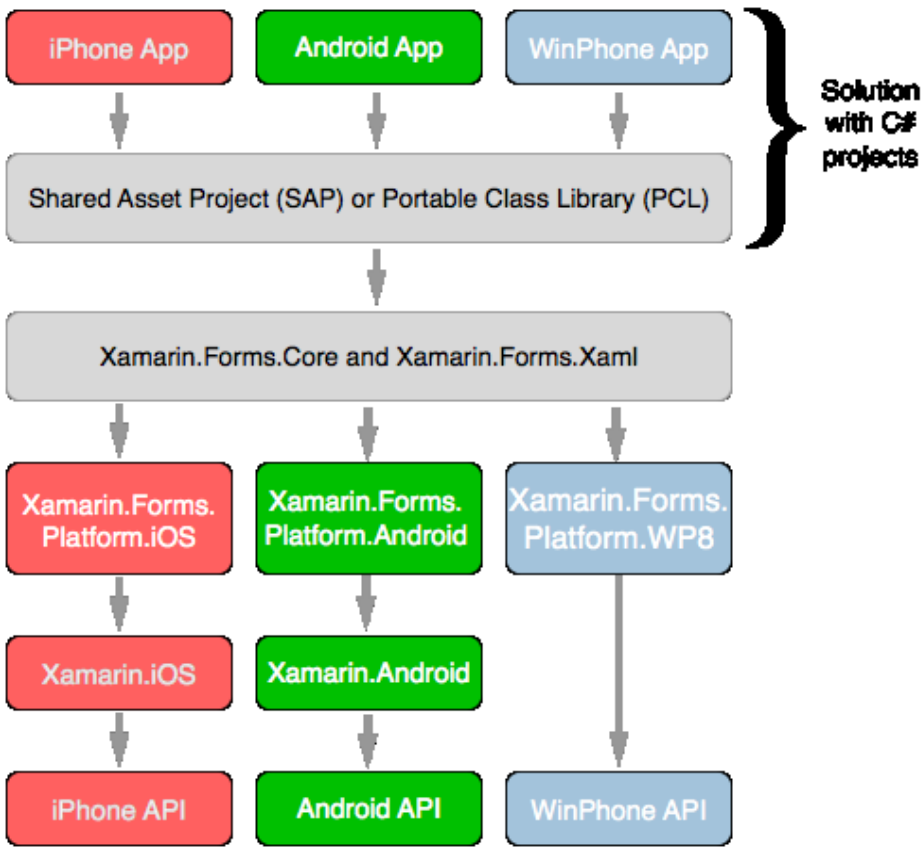
Introducing Xamarin.Forms

On May 28, 2014, Xamarin introduced Xamarin.Forms as part of a collection of enhancements to the Xamarin platform dubbed Xamarin 3. Xamarin.Forms allows you to write user-interface code that can be compiled for the iPhone, Android, and Windows Phone.

The Xamarin.Forms option

In the general case, a Xamarin.Forms application is three separate projects for the three mobile platforms with a fourth project containing common code—very much like the diagram that appeared in the previous section.

However, in a Xamarin.Forms application, the three platform projects are usually very small—often consisting of just stubs with a little boilerplate startup code. The Shared Asset Project, or the Portable Class Library project, contains the bulk of the application, including the user-interface code:



The Xamarin.Forms.Core and Xamarin.Forms.Xaml libraries implement the Xamarin.Forms API. Depending on the platform, Xamarin.Forms.Core then makes use of one of the Xamarin.Forms.Platform libraries. These libraries are mostly a collection of classes called renderers that transform the Xamarin.Forms user-interface objects into the platform-specific user interface.

The remainder of the diagram is the same as the one shown earlier.

For example, suppose you need the user-interface object discussed earlier that allows the user to toggle between two options. When programming for Xamarin.Forms, this is called a `Switch`, and a class named `Switch` is implemented in the Xamarin.Forms.Core library. In the individual renderers for the three platforms, this `Switch` is mapped to a `UISwitch` on the iPhone, a `Switch` on Android, and a `ToggleSwitchButton` on Windows Phone.

Xamarin.Forms.Core also contains a class named `Slider` for displaying a horizontal bar that the user manipulates to choose a numeric value. In the renderers in the platform-specific libraries, this is mapped to a `UISlider` on the iPhone, a `SeekBar` on Android, and a `Slider` on Windows Phone.

This means that when you write a Xamarin.Forms program that has a `Switch` or a `Slider`, what's actually displayed is the corresponding object implemented in each platform.

Here's a little Xamarin.Forms program containing a `Label` reading "Hello, Xamarin.Forms!", a `Button` saying "Click Me!", a `Switch`, and a `Slider`. The program is running on (from left to right) the iPhone, Android, and Windows Phone:

You'll see triple screenshots like this one throughout this book. They're always in the same order—iPhone, Android, and Windows Phone—and they're always running the same program.

As you can see, the `Button`, `Switch`, and `Slider` all have different appearances on the three phones because they are all rendered with the object specific to each platform.

What's even more interesting is the inclusion in this program of six `ToolBarItem` objects, three identified as primary items with icons, and three as secondary items without icons. On the iPhone these are rendered with `UIBarButtonItem` objects as the three icons and three buttons at the top of the page. On the Android, the first three are rendered as items on an `ActionBar`, also at the top of the page. On Windows Phone, they're realized as items on the `ApplicationBar` at the page's bottom.

The Android `ActionBar` has a vertical ellipsis and the Windows Phone `ApplicationBar` has a horizontal ellipsis. Tapping this ellipsis causes the secondary items to be displayed in a manner appropriate to these two platforms:

In one sense, Xamarin.Forms is an API that virtualizes the user-interface paradigms on each platform.

XAML support

Xamarin.Forms also supports XAML (pronounced "zammel" to rhyme with "camel"), the XML-based eXtensible Application Markup Language developed at Microsoft as a general-purpose markup language for instantiating and initializing objects. XAML isn't limited to defining initial layouts of user interfaces, but historically, that's what it's been used for the most, and that's what it's used for in Xamarin.Forms.

Here's the XAML file for the program whose screenshots you've just seen:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

```
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="PlatformVisuals.PlatformVisualsPage"
            Title="Visuals">

  <StackLayout Padding="10,0">
    <Label Text="Hello, Xamarin.Forms!"
           VerticalOptions="CenterAndExpand"
           HorizontalOptions="Center" />

    <Button Text = "Click Me!"
            VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center" />

    <Switch VerticalOptions="CenterAndExpand"
            HorizontalOptions="Center" />

    <Slider VerticalOptions="CenterAndExpand" />
  </StackLayout>

  <ContentPage.ToolbarItems>
    <ToolbarItem Name="edit" Order="Primary">
      <ToolbarItem.Icon>
        <OnPlatform x:TypeArguments="FileImageSource"
                    iOS="edit.png"
                    Android="ic_action_edit.png"
                    WinPhone="Images/edit.png" />
      </ToolbarItem.Icon>
    </ToolbarItem>

    <ToolbarItem Name="search" Order="Primary">
      <ToolbarItem.Icon>
        <OnPlatform x:TypeArguments="FileImageSource"
                    iOS="search.png"
                    Android="ic_action_search.png"
                    WinPhone="Images/feature.search.png" />
      </ToolbarItem.Icon>
    </ToolbarItem>

    <ToolbarItem Name="refresh" Order="Primary">
      <ToolbarItem.Icon>
        <OnPlatform x:TypeArguments="FileImageSource"
                    iOS="reload.png"
                    Android="ic_action_refresh.png"
                    WinPhone="Images/refresh.png" />
      </ToolbarItem.Icon>
    </ToolbarItem>

    <ToolbarItem Name="explore" Order="Secondary" />
    <ToolbarItem Name="discover" Order="Secondary" />
    <ToolbarItem Name="evolve" Order="Secondary" />
  </ContentPage.ToolbarItems>
</ContentPage>
```

Unless you have experience with XAML, some syntax details might be a little obscure. (Don't worry;

you'll learn all about them later on in this book.) But even so, you can see the `Label`, `Button`, `Switch`, and `Slider` tags. In a real program, the `Button`, `Switch`, and `Slider` would probably have event handlers attached that would be implemented in a C# code file. Here they do not. The `VerticalOptions` and `HorizontalOptions` attributes assist in layout; they are discussed in the next chapter.

Platform specificity

In the section of that XAML file involving the `ToolbarItem`, you can also see a tag named `OnPlatform`. This is one of several techniques in Xamarin.Forms that allow introducing some platform specificity in otherwise platform-independent code or markup. It's used here because each of the separate platforms has somewhat different image format and size requirements associated with these icons.

A similar facility exists in code with the `Device` class. It's possible to determine what platform the code is running on and to choose values or objects based on the platform. For example, you can specify different font sizes for each platform or run different blocks of code based on the platform. You might want to let the user manipulate a `Slider` to select a value in one platform but pick a number from a set of explicit values in another platform.

In some applications, deeper platform specificities might be desired. For example, suppose your application requires the GPS coordinates of the user's phone. This is not something that the initial version of Xamarin.Forms provides, so you'd need to write your own code specific to each platform to obtain this information.

The `DependencyService` class provides a way to do this in a structured manner. You define an interface with the methods you need (for example, `IGetCurrentLocation`) and then implement that interface with a class in each of the platform projects. You can then call the methods in that interface from the Xamarin.Forms project almost as easily as if it were part of the API.

As discussed earlier, each of the standard Xamarin.Forms visual objects—such as `Label`, `Button`, `Switch`, and `Slider`—are supported by a renderer class in the three Xamarin.Forms.Platform libraries. Each renderer class implements the platform-specific object that maps to the Xamarin.Forms object.

You can write your own custom visual objects with your own custom renderers. The custom visual object goes in the common code project, and the custom renderers go in the individual platform projects. To make it a bit easier, generally you'll want to derive from an existing class. Within the individual platform projects, all the corresponding renderers are public classes, and you can derive from them as well.

It's also possible to go in the other direction: You can build part of an application in the individual platform projects, and then make use of a page defined in the common Xamarin.Forms project. As soon as you begin writing Xamarin.Forms code in the next chapter of this book, you'll discover a method defined for the iPhone named `CreateViewController` that converts a Xamarin.Forms page into an iOS `UIViewController`. You'll discover an `AndroidActivity` class defined for the

Android that contains a `SetPage` method that accepts a Xamarin.Forms page. You'll see a method defined in Windows Phone called `ConvertPageToUIElement`.

Xamarin.Forms allows you to be as platform-independent or as platform-specific as you need to be. Xamarin.Forms doesn't replace Xamarin.iOS and Xamarin.Android; rather it integrates with them.

A cross-platform panacea?

For the most part, Xamarin.Forms focuses on areas of the mobile user interface that are common to the iOS, Android, and Windows Phone platforms to define its abstractions. These Xamarin.Forms visual objects are mapped to platform-specific objects, but Xamarin.Forms has tended to avoid implementing anything that is unique to a particular platform.

For this reason, despite the enormous help that Xamarin.Forms can offer in creating platform-independent applications, it is not a complete replacement for native API programming. If your application relies heavily on native API features such as particular types of controls or widgets, then you might want to stick with Xamarin.iOS, Xamarin.Android, and the native Windows Phone API—at least for the portions of the application that need this platform specificity.

You'll probably also want to stick with the native APIs for sections of your application that involve vector graphics or complex touch interaction. The initial version of Xamarin.Forms is not quite ready for these scenarios.

On the other hand, Xamarin.Forms is great for prototyping or making a quick proof-of-concept application. And after you've done that, you might just find that you can continue using Xamarin.Forms features to build the entire application. Xamarin.Forms is a natural for solid but not necessarily fancy line-of-business applications.

Even if you begin building the application with Xamarin.Forms and then implement major parts of it with platform APIs, you're doing so within a framework that allows you to share code and that offers structured ways to make platform-specific visuals.

Your development environment

How you set up your hardware and software depends on what mobile platforms you're targeting and what computing environments are most comfortable for you.

The requirements for Xamarin.Forms are no different from the requirements for using Xamarin.iOS or Xamarin.Android, or for programming for Windows Phone.

This means that nothing in this section (and the remainder of this chapter) is specific to Xamarin.Forms. There exists much documentation on the Xamarin website on setting up machines and software for Xamarin.iOS and Xamarin.Android programming, and on the Microsoft website about Windows Phone.

Machines and IDEs

If you want to target the iPhone, you're going to need a Mac. Apple requires that a Mac be used for building iPhone and other iOS applications. You'll need to install XCode on this machine and, of course, the Xamarin platform that includes the necessary libraries and Xamarin Studio. You can then use Xamarin Studio and Xamarin.Forms on the Mac for your iPhone development.

Once you have a Mac with XCode and the Xamarin platform installed, you can also install the Xamarin platform on a PC and program for the iPhone by using Visual Studio. The PC and Mac must be connected via a network (such as WiFi). You run the Xamarin.iOS Build Host on the Mac for this interconnection, and Visual Studio uses that to build and deploy the executable on the Mac.

Although you can run Xamarin Studio on the PC, you cannot use Xamarin Studio on the PC to do iPhone programming.

If you want to target Android phones, you have lots of flexibility. You can do so using Xamarin Studio on the Mac, Xamarin Studio on the PC, or Visual Studio on the PC.

If you want to target Windows Phone, you'll need to use Visual Studio 2012 or 2013 but not an Express edition.

This means that if you want to target all three platforms in a single IDE, you can do so with Visual Studio running on a PC connected to the Mac via a network. Another option is to run Visual Studio in a virtual machine on the Mac.

Devices and emulators

You can test your programs on a real phone connected to the machine via a USB cable, or you can test your programs with onscreen emulators.

There are advantages and disadvantages to each approach. A real phone is essential for testing complex touch interaction or when getting a feel for startup or response time. However, emulators allow you to see how your application adapts to a variety of sizes and form factors.

Perhaps the smoothest running emulator is that for the iPhone. However, because Mac desktop machines don't have touch screens, you'll need to use the mouse or mouse pad to simulate touch. The touch gestures on the Mac touch pad do not translate to the emulator. You can also connect a real iPhone to the Mac, but you'll need to provision it as a developer device.

The Windows Phone emulator is capable of several different screen resolutions and also tends to run fairly smoothly, albeit consuming lots of memory. If you run the Windows Phone emulator on a touch screen, you can use touch on the emulator screen. Connecting a real Windows Phone to the PC requires unlocking the phone. If you want to unlock more than one phone, you'll need a developer account.

Android emulators are the most problematic. They tend to be slow and cranky, although often extremely versatile in emulating a vast array of actual Android devices. On the up side, it's very easy to

connect a real Android phone to either the Mac or PC for testing. All you really need do is enable USB Debugging on the device.

If you like the idea of testing on real devices but you want to test on more real devices than you can possibly manage yourself, look into the Xamarin Test Cloud.

Installation

Before writing applications for Xamarin.Forms, you'll need to install the Xamarin platform on your Mac and/or your PC. See the articles on the Xamarin website at:

http://developer.xamarin.com/guides/cross-platform/getting_started/installation

You're probably eager to create your first Xamarin.Forms application, but before you do, you'll want to try creating normal Xamarin projects for the iPhone and Android, and a normal Windows Phone project.

This is important: If you're experiencing a problem using Xamarin.iOS or Xamarin.Android, that's not a problem with Xamarin.Forms, and you'll need to solve that problem before using Xamarin.Forms.

Creating an iOS app

If you're interested in using Xamarin.Forms to target the iPhone, first become familiar with the appropriate Getting Started documents on the Xamarin website:

http://developer.xamarin.com/guides/ios/getting_started/

This will give you guidance on using the Xamarin.iOS library to develop an iPhone application in C#. All you really need to do is get to the point where you can build and deploy a simple iPhone application on either a real iPhone or the iPhone simulator.

If you're using Xamarin Studio, you should be able to select **File > New > Solution** from the menu, and in the **New Solution** dialog, from the left select **C#** and **iOS** and then **Universal**, and from the template list in the center select **Empty Project**.

If you're using Visual Studio, you should be able to select **File > New > Project** from the menu, and in the **New Project** dialog, from the left select **Visual C#** and **iOS** and then **Universal**, and from the template list in the center select **Blank App (iOS)**.

In either case, select a location and name for the solution. Build and deploy the skeleton application created in the project. If you're having a problem with this, it's not a Xamarin.Forms issue. You might want to check the Xamarin.iOS forums to see if anybody else has a similar problem:

http://forums.xamarin.com/categories/ios

Creating an Android app

If you're interested in using Xamarin.Forms to target Android devices, first become familiar with the Getting Started documents on the Xamarin website:

http://developer.xamarin.com/guides/android/getting_started/

If you're using Xamarin Studio, you should be able to select **File > New > Solution** from the menu, and in the **New Solution** dialog, from the left select **C#** and **Android**, and in the template list in the center select **Android Application**.

If you're using Visual Studio, you should be able to select **File > New > Project** from the menu, and in the **New Project** dialog, from the left select **Visual C#** and then **Android**, and from the template list in the center select **Blank App (Android)**.

Give it a location and a name; build and deploy. If you can't get this process working, it's not a Xamarin.Forms issue, and you might want to check the Xamarin.Android forums for a similar problem:

http://forums.xamarin.com/categories/android

Creating a Windows Phone app

If you're interested in using Xamarin.Forms to target Windows Phone, you'll need to become familiar with at least the rudiments of using Visual Studio to develop Windows Phone applications:

http://dev.windows.com/

In Visual Studio 2013, you should be able select **File > New > Project** from the menu, and in the **New Project** dialog, at the left select **Visual C#**, then **Store Apps**, and **Windows Phone Apps**. In the center area, select the **Blank App (Windows Phone Silverlight)** template. From the dialog that follows, select **Windows Phone 8**.

You should be able to build and deploy the skeleton application to a real phone or an emulator. If not, search the Microsoft website or online forums such as Stack Overflow.

All ready?

If you can build Xamarin.iOS, Xamarin.Android, and Windows Phone applications (or some subset of those), then you're ready to create your first Xamarin.Forms application. It's time to say "Hello, Xamarin.Forms" to a new era in cross-platform mobile development.

CHAPTER 2

Pages, layouts, and views

The modern user interface is constructed from visual objects of various sorts. Depending on the operating system, these visual objects might go by different names—controls, elements, views, widgets—but they are all devoted to the jobs of presentation or interaction.

In Xamarin.Forms, the objects that appear on the screen are collectively called *visual elements*. They come in three main categories:

- *page*
- *layout*
- *view*

These are not abstract concepts! The Xamarin.Forms application programming interface (API) defines classes named `VisualElement`, `Page`, `Layout`, and `View`. These classes and their descendants form the backbone of the Xamarin.Forms user interface. `VisualElement` is an exceptionally important class in Xamarin.Forms. A `VisualElement` object is anything that occupies space on the screen.

A Xamarin.Forms application consists of one or more pages. A page usually occupies all (or at least a large area) of the screen. Some applications consist of only a single page, while others allow navigating among multiple pages. In this chapter you'll see just one type of page, called a `ContentPage`.

On each page, the visual elements are organized in a parent-child hierarchy. The child of a `ContentPage` is generally a layout of some sort to organize the visuals. Some layouts have a single child, but many layouts have multiple children that the layout arranges within itself. These children can be other layouts or views. Different types of layouts arrange children in a stack, or in a two-dimensional grid, or in a more freeform manner.

In this chapter, you'll encounter the `StackLayout` that arranges its children in a horizontal or vertical stack. You'll also see two types of layout that have a single child:

- `Frame` — displays a border around a child
- `ScrollView` — scrolls its child

The term *view* in Xamarin.Forms denotes familiar types of presentation and interactive objects: text, bitmaps, buttons, text-entry fields, sliders, switches, progress bars, date and time pickers, and others of your own devising. These are often called *controls* or *widgets* in other programming environments.

In this chapter, you'll see three types of views:

- `Label` — displays text
- `Button` — initiates commands
- `BoxView` — displays a simple colored box

Say hello

Using either Microsoft Visual Studio or Xamarin Studio, let's create a new Xamarin.Forms application using a standard template. This process actually creates a solution that contains up to four projects: three platform projects for iPhone, Android, and Windows Phone, and a common project for the greater part of your application code.

In Visual Studio, select the menu option **File > New > Project**. At the left of the **New Project** dialog, select **Visual C#** and then **Mobile Apps**.

In Xamarin Studio, select **File > New > Solution** from the menu, and at the left of the **New Solution** dialog, select **C#** and then **Mobile Apps**.

In either case, the center area of the dialog lists the available solution templates:

- **Blank App (Xamarin.Forms Portable)**
- **Blank App (Xamarin.Forms Shared)**
- **Class Library (Xamarin.Forms Portable)**

Now what? We definitely want to create a Blank App solution, but what kind?

The term "Portable" in this context refers to a Portable Class Library (PCL). All the common application code becomes a dynamic-link library (DLL) that is referenced by all the individual platform projects.

The term "Shared" in this context means a Shared Asset Project (SAP) containing loose code files (and perhaps other files) that are shared among the platform projects, essentially becoming part of each platform project.

For now, it doesn't matter which Blank App template you choose. Flip a coin if you're stymied. Select a disk location for the solution, and give it a name, for example, **Hello**.

If you're running Visual Studio, four projects are created: one common project (either a PCL or an SAP) and three application projects. For a solution named **Hello**, these are:

- A project named **Hello** that is common to all the application projects;
- An application project for Android, named **Hello.Android**;
- An application project for iOS, named **Hello.iOS**; and

- An application project for Windows Phone, named **Hello.WinPhone**.

If you're running Xamarin Studio on the Mac, the Windows Phone project isn't created, and if you're running Xamarin Studio on the PC, only the Android application project is created.

Before continuing, check to make sure the project configurations are OK. In Visual Studio, select the **Build > Configuration Manager** menu item. In the **Configuration Manager** dialog you'll see the three application projects. If you used the template that creates the Portable Class Library, that project is listed as well.

Make sure the **Build** box is checked for all the projects and the **Deploy** box is checked for all the application projects.

Take note of the **Platform** column: If the **Hello** project is listed, it should be flagged as **Any CPU**. The **Hello.Android** project should also be flagged as **Any CPU**. (For those two project types, **Any CPU** is the only option.) For the **Hello.iOS** project, choose either **iPhone** or **iPhoneSimulator** depending how you'll be testing the program. For the **Hello.WinPhone** project, you can select **x86** if you'll be using an on-screen emulator, **ARM** if you'll be deploying to a real phone, or **Any CPU** for deploying to either. Regardless of your choice, Visual Studio generates the same code.

In Xamarin.Studio on the Mac, you can switch between deploying to the iPhone and iPhone simulator through the **Project > Active Configuration** menu item.

In Visual Studio, you'll probably want to display the iOS and Android toolbars. These toolbars let you choose among emulators and devices, and allow managing the emulators. From the main menu, make sure the **View > Toolbars > iOS** and **View > Toolbars > Xamarin.Android** items are checked.

Because the solution contains anywhere from two to four projects, you must designate which program starts up when you elect to run or debug an application.

In the **Solution Explorer** of Visual Studio, right-click any of the application projects and select **Set As StartUp Project** item from the menu. You can then select to deploy to either an emulator or a real device. To build and run the program, select the menu item **Debug > Start Debugging**.

In the **Solution** list in Xamarin Studio, click the little gear icon that appears to the right of a selected project and select **Set as Startup Project** from the menu. You can then pick **Run > Start Debugging** from the main menu.

If all goes well, the skeleton application created by the template will run and you'll see a short message:

As you can see, these platforms have different color schemes. By default, the Windows Phone color scheme is like Android in that it displays light text on a dark background, but this color scheme is changeable by the user. Even on a Windows Phone emulator, you can change the color scheme in the **Themes** section of the **Settings** application, and then rerun the program.

The app is not only run on the device or emulator, but deployed. It shows up along with the other apps on the phone or emulator and can be run from there. If you don't like the application icon or how the app name shows up, you can change that in the individual platform projects.

Anatomy of a Xamarin.Forms solution

Obviously, the program created by the Xamarin.Forms template is very simple, so this is an excellent opportunity to examine the generated code files and figure out their interrelationships and how they work.

Let's begin with the code that's responsible for defining the text that you see on the screen. This is the App.cs file in the **Hello** project, and if the project template hasn't changed too much since this chapter was written, it probably looks something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using Xamarin.Forms;

namespace Hello
```

```
{
    public class App
    {
        public static Page GetMainPage()
        {
            return new ContentPage
            {
                Content = new Label {
                    Text = "Hello, Forms !",
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    HorizontalOptions = LayoutOptions.CenterAndExpand,
                },
            };
        }
    }
}
```

Notice that the namespace is the same as the project name. It is customary for the Xamarin.Forms `App` class in the PCL or SAP project to have a static `App.GetMainPage` method that returns a Xamarin.Forms `Page` object. The code that the Xamarin.Forms template has generated here shows one very simple approach to defining this method: The `ContentPage` class derives from `Page` and is very common in single-page Xamarin.Forms applications. (You'll see a lot of `ContentPage` throughout this book.) It occupies most of the phone's screen with the exception of standard icons at the top of the screen and buttons on the bottom of the Android screen.

The `ContentPage` class defines a property named `Content` that you set to the content of the page. Generally this content is a layout that in turn contains a bunch of views, but it could be a single view, as it is in this case.

The `Label` class derives from `View` and is used in Xamarin.Forms applications to display up to a paragraph of text. The `VerticalOptions` and `HorizontalOptions` properties are discussed in more detail later in this chapter.

For your own single-page Xamarin.Forms applications, you'll generally be defining your own class that derives from `ContentPage`. The `App.GetMainPage` class then returns an instance of the class you defined. You'll see an example shortly.

If you chose the **Blank App (Xamarin.Forms Shared)** template, there is nothing else of much interest in the **Hello** project other than the App.cs file. If you used the **Blank App (Xamarin.Forms Portable)** template, you'll see an AssemblyInfo.cs file necessary for creating the PCL. In the **References** section under **Hello** in the solution list, you'll see the three libraries this PCL requires:

- .NET
- Xamarin.Forms.Core
- Xamarin.Forms.Xaml

It is this common project that will receive the bulk of your attention as you're writing a Xamarin.Forms application. In some circumstances the code in this project might require some tailoring

for the three different platforms, and you'll see shortly how to do that. You can also include platform-specific code in the three application projects.

The three application projects have their own assets in the form of icons and metadata, and you must pay these assets particular attention if you intend to bring the application to market. But during the time that you're learning how to develop applications using Xamarin.Forms, these assets can generally be ignored. You'll probably want to keep these application projects collapsed in the solution list because you don't need to bother much with their contents.

But you really should know what's in these application projects, so let's take a closer look.

In the **References** section of each application project, you'll see references to the common project (**Hello** in this case), as well as various .NET assemblies, the Xamarin.Forms assembles listed above, and additional Xamarin.Forms assemblies applicable to each platform:

- Xamarin.Forms.Platform.Android
- Xamarin.Forms.Platform.iOS
- Xamarin.Forms.Platform.WP8

Each of these three libraries defines a static `Xamarin.Forms.Init` method to initialize the Xamarin.Forms system. You've also just seen that the common project customarily defines a public static method named `App.GetMainPage`,

At the very least, each of the three application projects must make calls to these two methods in the context of platform-specific overhead. If you're familiar with iOS, Android, or Windows Phone development, you might be curious to discover what each of these three platforms does with the Xamarin.Forms `Page` object returned from that `App.GetMainPage` method.

The iOS project

An iOS project typically contains a class that derives from `UIApplicationDelegate`. The standard Xamarin.Forms template defines such a class, which can be found in the AppDelegate.cs file in the Hello.iOS project. Here it is stripped of all extraneous `using` directives and comments:

```
using MonoTouch.Foundation;
using MonoTouch.UIKit;
using Xamarin.Forms;

namespace Hello.iOS
{
    [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.GetMainPage().CreateViewController();
            window.MakeKeyAndVisible();

            return true;
        }
    }
}
```

The `FinishedLaunching` override begins by calling the `Forms.Init` method defined in the Xamarin.Forms.Platform.iOS assembly. It then creates a new `UIWindow` instance (as usual) but calls the static `App.GetMainPage` method in the common **Hello** project and uses an extension method named `CreateViewController` (defined in the Xamarin.Forms.Platform.iOS assembly) to convert that page into an object of type `UIViewController`.

The object that `CreateViewController` returns is a class private to the Xamarin.Forms.Platform.iOS assembly named `PlatformRenderer` that is responsible for rendering the page's contents.

The Android project

In the Android application, the typical `MainActivity` class must be derived from a Xamarin.Forms class named `AndroidActivity` defined in the Xamarin.Forms.Platform.Android assembly, and the `Forms.Init` call requires some additional information:

```
using Android.App;
using Android.Content.PM;
using Android.OS;

using Xamarin.Forms.Platform.Android;

namespace Hello.Droid
{
    [Activity(Label = "Hello", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : AndroidActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            Xamarin.Forms.Forms.Init(this, bundle);

            SetPage(App.GetMainPage());
        }
    }
}
```

The return value from the `App.GetMainPage` call is passed directly to the `SetPage` method defined

by the `AndroidActivity` class and inherited by `MainActivity`.

The Windows Phone project

In the Windows Phone project, look at the MainPage.xaml.cs file tucked underneath the MainPage.xaml file in the project file list. This file defines the customary `MainPage` class and uses an extension method defined in the Xamarin.Forms.Platform.WP8 assembly named `ConvertPageToUIElement` on the object return from `App.GetMainPage`:

```
using Microsoft.Phone.Controls;
using Xamarin.Forms;

namespace Hello.WinPhone
{
    public partial class MainPage : PhoneApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();

            Forms.Init();
            Content = Hello.App.GetMainPage().ConvertPageToUIElement(this);
        }
    }
}
```

If you're a Windows Phone programmer, you might be interested to know that `ConvertPageToUIElement` returns a Windows Phone `Canvas` object.

You might want to add the following statement to the `MainPage` constructor right after `InitializeComponent`:

```
this.SupportedOrientations = SupportedPageOrientation.PortraitOrLandscape;
```

That will allow the Windows Phone display to respond to orientation changes and be consistent with the iPhone and Android projects.

Nothing special!

If you've created a Xamarin.Forms solution under Visual Studio and don't wish to target one or more platforms, simply delete those projects.

If you later change your mind about those projects—or you originally created the solution in Xamarin Studio and you want to move it to Visual Studio to target Windows Phone—you can add new platform projects to the Xamarin.Forms solution. In the **Add New Project** dialog, you can create a Xamarin.iOS project by selecting the iOS project **Universal** type and **Blank App** template, or a Xamarin.Android project with the Android **Blank App** template, or a Windows Phone project with the **Store Apps – Windows Phone Apps - Blank App (Windows Phone Silverlight)** template.

For these new projects you can get the correct references and boilerplate code by consulting the projects generated by the standard Xamarin.Forms template.

To summarize: There's really nothing all that special in a Xamarin.Forms app compared with normal Xamarin or Windows Phones projects—except the Xamarin.Forms libraries.

PCL or SAP?

When you first created the **Hello** solution, you had a choice of two application templates:

- **Blank App (Xamarin.Forms Portable)**
- **Blank App (Xamarin.Forms Shared)**

The first creates a Portable Class Library (PCL) while the second creates a Shared Asset Project (SAP) consisting only of shared code files.

In both cases, this project becomes part of the three applications, but in decidedly different ways: With the PCL approach, all the common code is bundled into a dynamic-link library that each application project references and binds to at run time.

With the Shared Asset Project approach, the common code files are effectively included with the three application projects at build time. As you saw, by default the SAP has only a single file—the App.cs file described above. It's exactly as if this project did not exist and instead there were three different copies of this file in the three application projects.

Some subtle (and not-so-subtle) problems can manifest themselves with the **Blank App (Xamarin.Forms Shared)** template:

The iOS and Android projects have access to pretty much the same version of .NET, but it is not the same version of .NET that the Windows Phone project uses. This means that any .NET classes accessed by the shared code might be somewhat different depending on the platform. As you'll discover in the next chapter, this is the case for some file I/O classes in the `System.IO` namespace.

You can compensate for these differences by using C# preprocessor directives, particularly `#if` and `#elif`. In the projects generated by the Xamarin.Forms template, the Windows Phone and iPhone projects define symbols that you can use with these directives.

What are these symbols?

In Visual Studio, right-click the project name in the **Solution Explorer** and select **Properties**. At the left of the properties screen, select **Build**, and look for the **Conditional compilation symbols** field.

In Xamarin.Studio, select an application project in the **Solution** list, invoke the dropdown tools menu, and select **Options**. In the left of the **Project Options** dialog, select **Build > Compiler**, and look for the **Define Symbols** field.

You discover that the symbol `__IOS__` is defined for the iPhone project (that's two underscores

before and after) and `WINDOWS_PHONE` is defined for the Windows Phone project. Nothing special is defined for Android, but that's OK, because your shared code file can include blocks like this:

```
#if __IOS__
            // iPhone specific code
#elif WINDOWS_PHONE
            // Windows Phone specific code
#else
            // Android specific code
#endif
```

This allows your shared code files to run platform-specific code or access platform-specific classes, including classes in the individual platform projects. You can also define your own conditional compilation symbols if you'd like.

These preprocessor directives make no sense in a Portable Class Library project. The PCL is entirely independent of the three platforms and these identifiers in the platform projects are ignored when the PCL is compiled.

The concept of the PCL originally arose because every platform that uses .NET actually uses a somewhat different subset of .NET. If you want to create a library that can be used among multiple .NET platforms, you need to use only the common parts of those .NET subsets.

The PCL is intended to help by containing code that is usable on multiple (but specific) .NET platforms. Consequently, any particular PCL contains some embedded flags that indicate what platforms it supports. A PCL used in a Xamarin.Forms application must support the following platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone Silverlight 8
- Xamarin.Android
- Xamarin.iOS

If you need platform-specific behavior in the PCL—you'll see an example shortly—you can't use the C# preprocessor directives because those only work at build time. You need something that works at run time, such as the Xamarin.Forms `Device` class. However, the PCL cannot reference code that can't be compiled into the PCL.

The Xamarin.Forms PCL can access other PCLs supporting the same platforms, but it cannot directly access classes defined in the individual application projects. However, if that's something you need to do—and you'll see an example in the next chapter—Xamarin.Forms provides a class named `DependencyService` that allows you to access platform-specific code from the PCL in a methodical manner.

In your early days of creating Xamarin.Forms projects, you might want to bounce back and forth

between the PCL and SAP approaches just to get a feel for them. However, once you begin incorporating XAML in your applications, you'll discover that XAML files must be in a PCL. For purposes of consistency, most of the projects in this book use the PCL approach. Programmers who have been working with Xamarin.Forms for a while generally prefer PCL, but SAP definitely has its advocates too.

But why choose? You can have both in the same solution. If you've created a Xamarin.Forms solution with a Shared Asset Project, you can add a new PCL project to the solution by selecting the **Class Library (Xamarin.Forms Portable)** template. The application projects can access both the SAP and PCL, and the SAP can access the PCL as well.

Labels for text

Let's create a new Xamarin.Forms solution, named **Greetings**, using the same process described above for creating the **Hello** solution. This new solution will be structured more like a typical Xamarin.Forms program, which means that it will define a new class that derives from `ContentPage`. Most of the time in this book, every class and structure defined by a program will get its own file. This means that a new file must be added to the **Greetings** project:

In Visual Studio, you can right click the **Greetings** project in the **Solution Explorer** and select **Add > New Item** from the menu. At the left of the **Add New Item** dialog, select **Visual C#** and **Code**, and in the center area, select **Class**.

In Xamarin Studio, from the tool icon on the **Greetings** project select **Add > New File** from the menu. In the left of the **New File** dialog, select **General**, and in the central area, select **Empty Class**.

In either case, give it a name of GreetingsPage.cs.

The GreetingsPage.cs file will be initialized with some skeleton code for a class named **GreetingsPage**. You'll want this class to derive from the Xamarin.Forms class `ContentPage`, so you'll need a `using` directive for the Xamarin.Forms namespace, and you'll need to specify that your class derives from `ContentPage`. The `GreetingsPage` class does not need to be public because it won't be directly accessed from outside the **Greetings** project, but you will want a parameterless constructor. Here's the empty class ready for some code:

```
using System;
using Xamarin.Forms;

namespace Greetings
{
    class GreetingsPage : ContentPage
    {
        public GreetingsPage()
        {

        }
    }
```

```
}
```

In the constructor of the `GreetingsPage` class, create a `Label` view, set its `Text` property, and set that `Label` instance to the `Content` property that `GreetingsPage` inherits from `ContentPage`:

```
using System;
using Xamarin.Forms;

namespace Greetings
{
    class GreetingsPage : ContentPage
    {
        public GreetingsPage()
        {
            Label label = new Label();
            label.Text = "Greetings, Xamarin.Forms!";
            this.Content = label;
        }
    }
}
```

Now change the `App` class in App.cs to return an instance of this `GreetingsPage` class:

```
using System;
using Xamarin.Forms;

namespace Greetings
{
    public class App
    {
        public static Page GetMainPage()
        {
            return new GreetingsPage();
        }
    }
}
```

It is the `GreetingsPage` class (and others like it) where you'll be spending most of your time in early Xamarin.Forms programming. For some single-page UI-intensive programs, this class might contain the only application code that you'll need to write. Of course, you can add additional classes to the project if you need them.

In many of the single-page sample programs in this chapter, the class that derives from `ContentPage` will have a name that is the same as the application but with the word `Page` appended. That naming convention should allow you to easily identify the project name from the class or constructor name without seeing the entire file. In most cases, the code snippets in these pages won't include the `using` directives or the namespace definition.

Many Xamarin.Forms programmers prefer to use the C# 3.0 style of object creation and property initialization in their page constructors. Following the `Label` constructor, a pair of curly braces enclose one or more property settings separated by commas. Here's an alternative (but functionally equivalent)

`GreetingsPage` definition:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        Label label = new Label
        {
            Text = "Greetings, Xamarin.Forms!"
        };
        this.Content = label;
    }
}
```

This style allows the `Label` instance to be set directly to the `Content` property directly, like so:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!"
        };
    }
}
```

For more complex page layouts, this style of instantiation and initialization provides a better visual analogue of the organization of layouts and views on the page. However, it's not always as simple as this example might indicate if you need to call methods on these objects or set event handlers.

Whichever way you do it, if you can successfully run the program on the three platforms in either an emulator or a device, here's what you'll see:

The most disappointing version of this **Greetings** program is definitely the iPhone: In iOS 7, a single-page application shares the screen with the status bar at the top. Anything the application displays at the top of its page will occupy the same space as the status bar unless the application compensates for it.

This problem disappears in multipage navigation applications, but until that time, here are four ways to solve this problem right away:

1. Include padding on the page

The `Page` class defines a property named `Padding` that marks an area around the interior perimeter of the page into which content cannot intrude. The `Padding` property is of type `Thickness`, a structure that defines four properties named `Left`, `Top`, `Right`, `Bottom`. (You might want to memorize that order because that's the order you'll define them in the `Thickness` constructor as well as in XAML.) The `Thickness` structure also defines constructors for setting the same amount of padding on all four sides or for setting the same amount on the left and right and on the top and bottom.

A little research in your favorite search engine will reveal that the iOS 7 status bar has a height of 20. (Twenty what? you might ask. Twenty pixels? Actually no. For now, just think of them as 20 "units." For most Xamarin.Forms programming you shouldn't need to bother with numeric sizes, but Chapter 5 will provide some guidance when you need to get down to the pixel level.) You can accommodate the status bar like so:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
```

```
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!"
        };

        this.Padding = new Thickness(0, 20, 0, 0);
    }
}
```

Now the greeting appears 20 units from the top of the page:

2. Include padding just for iOS

Setting the `Padding` property on the `ContentPage` solves the problem of the text overwriting the iOS status bar, but it also sets the same padding on the Android and Windows Phone where it's not required. Is there a way to set this padding only on the iPhone?

Yes! The static `Device` class includes several properties and methods that allow your code to deal with device differences in a very simple and straightforward manner:

- The `Device.OS` property returns a member of the `TargetPlatform` enumeration: `iOS`, `Android`, `WinPhone`, or `Other`.
- The `Device.Idiom` property returns a member of the `TargetIdiom` enumeration: `Phone`, `Tablet`, `Desktop`, or `Unsupported`. (However, Xamarin.Forms is really only intended for phones at this time.)

You can use these two properties in `if` and `else` statements to execute code specific to a

particular platform.

Two methods named `OnPlatform` provide even more elegant solutions:

- The static generic method `OnPlatform<T>` takes three arguments of type `T`, the first for iOS, the second for Android, and the third for Windows Phone, and returns the argument for the running platform.
- The static method `OnPlatform` has four arguments of type `Action` (the .NET function delegate that has no arguments and returns `void`) also in the order iOS, Android, and Windows Phone, with a fourth for a default, and executes the argument for the running platform.

The `Device` class also defines a static method to start running a timer, to invoke the phone's email or telephone dialer, and to run some code on the main thread. This latter method (called `Device.BeginInvokeOnMainThread`) comes in handy when you're working with asynchronous methods and supplemental threads because code that manipulates the user interface can generally only be run on the main user-interface thread.

Rather than setting the same `Padding` property on all three platforms, you can restrict the `Padding` to just the iPhone by using the `Device.OnPlatform` generic method:

```
double topPadding = Device.OnPlatform<double>(20, 0, 0);
this.Padding = new Thickness(0, topPadding, 0, 0);
```

Explicitly specifying the type of the `Device.OnPlatform` arguments within the angle brackets isn't required if the compiler can figure it out from the arguments. In this case, if the `double` in angle brackets is removed, the compiler would determine that the arguments are integers and then implicitly convert the integer to a `double` in the assignment statement.

You can simplify this code to just:

```
this.Padding = new Thickness (0, Device.OnPlatform (20, 0, 0), 0, 0);
```

This is how the `Padding` will be set in the programs that follow when it's required. Of course, you can substitute some other numbers for the 0's if you want some additional padding on the page. Sometimes a little padding on the sides makes for a more attractive display.

In Shared Asset Projects you can use the C# preprocessor directives `#if` and `#elif` as shown earlier, but the `Device` class has the advantage of working with both PCL projects and Shared Asset Projects.

3. Center the label within the page

The problem with the text overlapping the iOS status bar only occurs because the text by default is displayed at the upper-left corner. Is it possible to center the text on the page?

Xamarin.Forms supports a number of facilities to ease layout without requiring the program to perform calculations involving sizes and coordinates. The `View` class defines two properties named

`HorizontalOptions` and `VerticalOptions` that specify how a view is to be positioned relative to its parent. These two properties are both of type `LayoutOptions`, a structure with eight public static read-only fields:

- `Start`
- `Center`
- `End`
- `Fill`
- `StartAndExpand`
- `CenterAndExpand`
- `EndAndExpand`
- `FillAndExpand`

The `LayoutOptions` structure also defines two instance properties that let you formulate these same combinations:

- An `Alignment` property of type `LayoutAlignment`, an enumeration with four members: `Start`, `Center`, `End`, and `Fill`.
- An `Expands` property of type `bool`.

You can set the `HorizontalOptions` and `VerticalOptions` properties defined by `View` to a `LayoutOptions` value. For `HorizontalOptions`, the word `Start` means left and `End` means right; for `VerticalOptions`, `Start` means top and `End` means bottom.

Mastering the use of `HorizontalOptions` and `VerticalOptions` properties is a major part of acquiring skill in the Xamarin.Forms layout system, but here's a simple example. This third solution to the iOS status bar problem positions the `Label` in the center of the page:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
    }
}
```

Here's how it looks:

This is the version of the **Greetings** program that is included in the sample code for this chapter. You can use various combinations of `HorizontalOptions` and `VerticalOptions` to position the text in any of nine places relative to the screen.

You'll be learning much more about `HorizontalOptions` and `VerticalOptions` properties and the `LayoutOptions` structure in the pages ahead.

4. Center the text within the label

The `Label` is intended to display text up to a paragraph in length. It is often desirable to control how the lines of text are horizontally aligned: left-justified, right-justified, or centered.

The `Label` view defines an `XAlign` property for that purpose and also a `YAlign` property for positioning text vertically. Both properties are set to a member of the `TextAlignment` enumeration, which has members named `Start`, `Center`, and `End` to be versatile enough for text that runs from right to left or from top to bottom.

For English and other European languages, `Start` means left or top, and `End` means right or bottom.

For this fourth solution to the iOS status bar problem, set `XAlign` and `YAlign` to `TextAlignment.Center`.

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
```

```
        {
            Text = "Greetings, Xamarin.Forms!",
            XAlign = TextAlignment.Center,
            YAlign = TextAlignment.Center
        };
    }

}
```

Visually, the result with this single line of text is the same as setting `HorizontalOptions` and `VerticalOptions` to center, and you can also use various combinations of these properties to position the text in one of nine different locations around the page.

However, these two techniques to center the text are actually quite different, as you'll soon see.

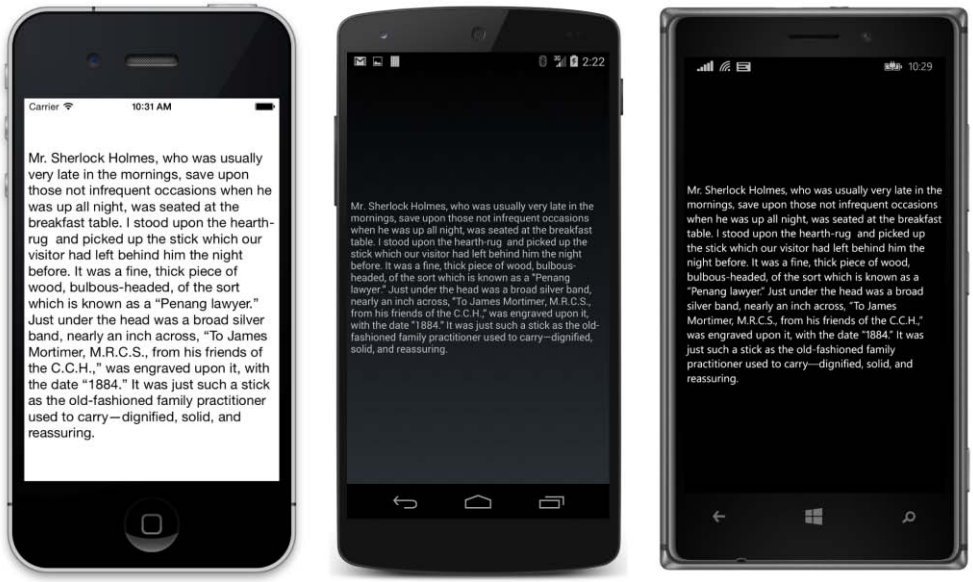
Wrapping paragraphs

Displaying a paragraph of text is as easy as displaying a single line of text:

```
class BaskervillesPage : ContentPage
{
    public BaskervillesPage()
    {
        this.Content = new Label
        {
            VerticalOptions = LayoutOptions.Center,
            Text =
                "Mr. Sherlock Holmes, who was usually very late in " +
                "the mornings, save upon those not infrequent " +
                "occasions when he was up all night, was seated at " +
                "the breakfast table. I stood upon the hearth-rug  " +
                "and picked up the stick which our visitor had left " +
                "behind him the night before. It was a fine, thick " +
                "piece of wood, bulbous-headed, of the sort which " +
                "is known as a \u201CPenang lawyer.\u201D Just " +
                "under the head was a broad silver band, nearly an " +
                "inch across, \u201CTo James Mortimer, M.R.C.S., " +
                "from his friends of the C.C.H.,\u201D was engraved " +
                "upon it, with the date \u201C1884.\u201D It was " +
                "just such a stick as the old-fashioned family " +
                "practitioner used to carry\u2014dignified, solid, " +
                "and reassuring."
        };

        this.Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);
    }
}
```

Notice the use of embedded Unicode codes for opened and closed "smart quotes" (`\u201C` and `\x201D`) and the em-dash (`\u2014`). Padding has been set for 5 units around the page to avoid the text butting up against the edges of the screen, but the `VerticalOptions` property has been used

as well to vertically center the entire paragraph on the page:



For this paragraph of text, setting `HorizontalOptions` will shift the entire paragraph horizontally slightly to the left, center, or right. The shifting is only slight because the width of the paragraph is the width of the longest line of text. Since word wrapping is governed by the page width (minus the padding), the paragraph likely occupies slightly less width than the width available for it on the page.

But setting `XAlign` has a much more profound affect: This affects the alignment of the individual lines. A setting of `TextAlignment.Center` will center all the lines of the paragraph, and `TextAlignment.Right` aligns them all at the right. You can use `HorizontalOptions` in addition to `XAlign` to shift the entire paragraph slightly to the center or right.

However, after you've set `VerticalOptions` to `Start`, `Center`, or `End`, `YAlign` has no effect.

`Label` defines a `LineBreakMode` property that you can set to a member of the `LineBreakMode` enumeration if you don't want the text to wrap, or to select truncation options.

There is no property to specify a first-line indent for the paragraph but you can add one of your own with space characters of various types, such as the em-space (Unicode `\u2003`).

You can display multiple paragraphs with a single `Label` view by ending each paragraph with one or more line feed characters (`\n`). However, it makes more sense to use a separate `Label` view for each paragraph, as will be demonstrated shortly.

The `Label` class has lots of formatting flexibility. As you'll see shortly, a `Font` property lets you select a font size, or bold or italic text, and you can also specify different text formatting within a single paragraph.

`Label` also allows specifying color, and a little experimentation with color will demonstrate the profound difference between `HorizontalOptions` and `VerticalOptions`, and `XAlign` and `YAlign`.

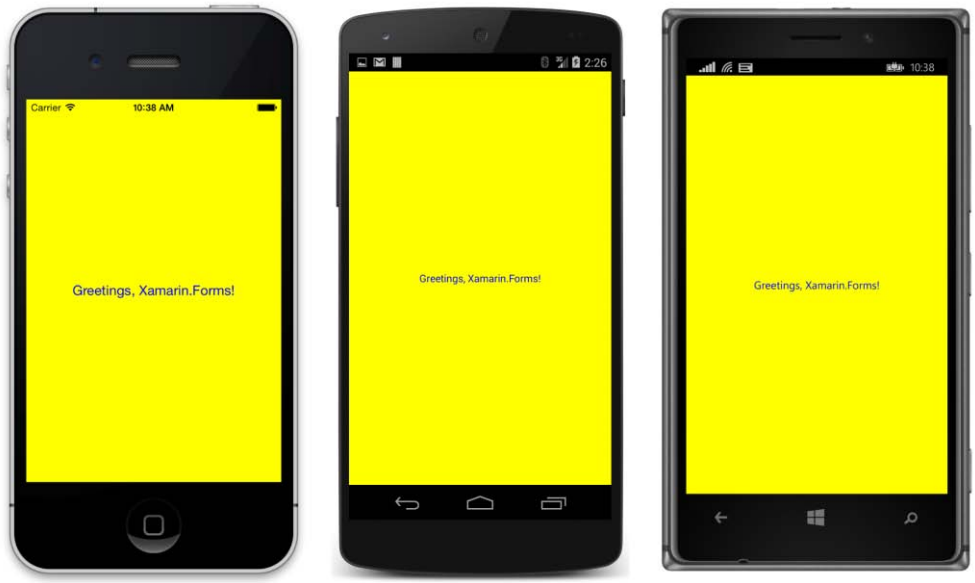
Text and background colors

As you've seen, the `Label` view displays text in a color appropriate for the device. You can override that behavior by setting two properties named `TextColor` and `BackgroundColor`. `Label` itself defines `TextColor` but inherits `BackgroundColor` from `VisualElement`, which means that `Page` and `Layout` also have a `BackgroundColor` property.

You set `TextColor` and `BackgroundColor` to a value of type `Color`, which is a structure that defines 16 static fields for obtaining common colors. Here are two of them used in conjunction with `XAlign` and `YAlign` to center the text:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            XAlign = TextAlignment.Center,
            YAlign = TextAlignment.Center,
            BackgroundColor = Color.Yellow,
            TextColor = Color.Blue
        };
    }
}
```

The result might surprise you. As these screenshots illustrate, the `Label` actually occupies the entire area of the page (including underneath the iOS status bar), and the `XAlign` and `YAlign` properties position the text within that entire area:



Here's some code that colors the text the same but instead centers the text using the `HorizontalOptions` and `VerticalOptions` properties:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            BackgroundColor = Color.Yellow,
            TextColor = Color.Blue
        };
    }
}
```

Now the `Label` occupies only as much space as required for the text, and that's what's positioned in the center of the page:

The default values of `HorizontalOptions` and `VerticalOptions` are not `LayoutOptions.Start` as the default appearance of the text might suggest. The default values are instead `LayoutOptions.Fill`. This is the setting that causes the `Label` to fill the page. The default `XAlign` and `YAlign` values of `TextAlignment.Start` are what caused the text to be positioned at the upper-left in the first version of the **Greetings** program.

You might wonder: What are the default values of the `TextColor` and `BackgroundColor` properties, because the default values result in different colors for the different platforms?

The default values of `TextColor` and `BackgroundColor` are actually a special color value named `Color.Default` that does not represent a real color but instead is used to reference the text and background colors appropriate for the particular platform.

Let's explore color in more detail.

The Color structure

Internally, the `Color` structure stores colors in two different ways:

- As red, green, and blue (RGB) values of type `double` that range from 0 to 1. Read-only properties named `R`, `G`, and `B` expose these values.
- As hue, saturation, and luminosity values of type `double` that also range from 0 to 1. These values are exposed with read-only properties named `Hue`, `Saturation`, and `Luminosity`.

The `Color` structure also supports an alpha channel for indicating degrees of opacity. A read-only

property named `A` exposes this value, which ranges from 0 for transparent to 1 for opaque. All these properties are read-only. Once created, a `Color` value is immutable.

You can create a `Color` value in one of several ways. The three constructors are the easiest:

- `new Color(double grayShade)`
- `new Color(double r, double g, double b)`
- `new Color(double r, double g, double b, double a)`

Arguments can range from 0 to 1. `Color` also defines several static creation methods, including:

- `Color.FromRgb(double r, double g, double b)`
- `Color.FromRgb(int r, int g, int b)`
- `Color.FromRgba(double r, double g, double b, double a)`
- `Color.FromRgba(int r, int g, int b, int a)`
- `Color.FromHsla(double h, double s, double l, double a)`

The two static methods with integer arguments assume that the values range from 0 to 255, which is the customary representation of RGB colors. Internally, the constructor simply divides the integer values by 255.0 to convert to `double`.

Watch out! You might think that you're creating a red color with this call:

```
Color.FromRgb(1, 0, 0)
```

However, the C# compiler will assume that these arguments are integers. The integer method will be invoked, and the first argument will be divided by 255.0, with a result that is nearly zero. If you want the method that has `double` arguments, be explicit:

```
Color.FromRgb(1.0, 0, 0)
```

`Color` also defines static creation methods for a packed `uint` format and a hexadecimal format in a string.

The `Color` structure also defines 16 public static read-only fields of type `Color`. Here they are shown with the integer RGB values that the `Color` structure uses internally to define these fields, and the corresponding `Hue`, `Saturation`, and `Luminosity` values, somewhat rounded for purposes of clarity:

Color Fields	Red	Green	Blue	Hue	Saturation	Luminosity
White	255	255	255	0	0	1.00
Silver	192	192	192	0	0	0.75
Gray	128	128	128	0	0	0.50
Black	0	0	0	0	0	0
Red	255	0	0	1.00	1	0.50
Maroon	128	0	0	1.00	1	0.25

Yellow	255	255	0	0.17	1	0.50
Olive	128	128	0	0.17	1	0.25
Lime	0	255	0	0.33	1	0.50
Green	0	128	0	0.33	1	0.25
Aqua	0	255	255	0.50	1	0.50
Teal	0	128	128	0.50	1	0.25
Blue	0	0	255	0.67	1	0.50
Navy	0	0	128	0.67	1	0.25
Fuschia	255	0	255	0.83	1	0.50
Purple	128	0	128	0.83	1	0.25

You might recognize these as the color names supported in HTML. A 17th public static read-only field is named `Transparent`, which has `R`, `G`, `B`, and `A` properties all set to zero.

When people are given an opportunity to interactively formulate a color, the HSL color model is often more intuitive than RGB. The `Hue` cycles through the colors of the visible spectrum (and the rainbow) beginning with red at 0, green at 0.33, blue at 0.67, and back to red at 1.

The `Saturation` indicates the degree of the hue in the color, ranging from 0, which is no hue at all and results in a gray shade, to 1 for full saturation.

The `Luminosity` is a measure of lightness, ranging from 0 for black to 1 for white.

In Chapter 4, a color-selection program lets you explore the RGB and HSL models more interactively.

The `Color` structure includes several interesting instance methods that allow creating new colors that are modifications of existing colors:

- `AddLuminosity(double delta)`
- `MultiplyAlpha(double alpha)`
- `WithHue(double newHue)`
- `WithLuminosity(double newLuminosity)`
- `WithSaturation(double newSaturation)`

Finally, `Color` defines two special static read-only properties of type `Color`:

- `Color.Default`
- `Color.Accent`

The `Color.Default` property is used extensively in defining Xamarin.Forms default color values. The `VisualElement` class initializes its `BackgroundColor` property as `Color.Default`, and the `Label` class initializes its `TextColor` property as `Color.Default`.

However, `Color.Default` is a `Color` value with its `R`, `G`, `B`, and `A` properties all set to –1, which means that it's a special "mock" value that means nothing in itself, but indicates that the actual value is platform-specific.

For `Label` and `ContentPage` (and most classes that derive from `VisualElement`), the `BackgroundColor` setting of `Color.Default` means transparent.

When you set the `TextColor` property to `Color.Default`, this means black on an iOS device, white on an Android device, and either white or black on Windows Phone, depending on the color theme selected by the user.

Without digging into platform-specific user-interface objects, a Xamarin.Forms program cannot determine if the underlying color scheme is white-on-black or black-on-white. This can be a little frustrating if you want to create colors that are compatible with the color scheme, for example a dark blue text color if the default background is white, or light yellow text if the default background is dark.

You have a couple of strategies for working with color: You can choose to do your Xamarin.Forms programming in a very platform-independent manner and avoid making any assumptions about the default color scheme of any phone. Or, you can use your knowledge about the color schemes of the various platforms and use `Device.OnPlatform` to specify platform-specific colors.

But don't try to just ignore all the defaults and the platforms and explicitly set all the colors in your application to your own color scheme. This probably won't work as well as you hope, because many views use other colors that relate to the color theme of the operating system but that are not exposed through Xamarin.Forms properties.

One easy option is to use the `Color.Accent` property for an alternative text color. On the iPhone and Android, it's a color that is visible against the default background, but it's not the default text color. On the Windows Phone, it's a color selected by the user as part of the color theme.

You can make text semi-transparent by setting `TextColor` to a `Color` value with an `A` property less than 1. However, if you want a semi-transparent version of the default text color, use the `Opacity` property of the `Label` instead. This property is defined by the `VisualElement` class and has a default value of 1. Set it to values less than 1 for various degrees of transparency.

Standard fonts and sizes

The `Label` uses a default (or system) font defined by each platform, but `Label` also defines a `Font` property that you can use to change this font. `Label` is only one of two classes with a `Font` property; `Button` is the other.

The `Font` property is of type `Font`, a structure with several static methods that create `Font` values. These `Font` values are immutable. The current recommended `Font` creation methods are named:

- `Font.SystemFontOfSize`
- `Font.OfSize`

These two methods both require an argument specifying the size of the desired font in one of two

ways—as a number or as a member of the `NamedSize` enumeration:

- `NamedSize.Micro`
- `NamedSize.Small`
- `NamedSize.Medium` (the default size)
- `NamedSize.Large`

Consequently, you can use the `Font` class in easy ways or hard ways:

One hard way is to use the `Font.OfSize` method. This method allows you to choose a specific font family in the form of a text string, but the available fonts are different on each of the three platforms, so you must know something about these available fonts to successfully use this method. For this reason, a fuller discussion of the `Font.OfSize` method will be delayed until a later chapter not in this Preview Edition.

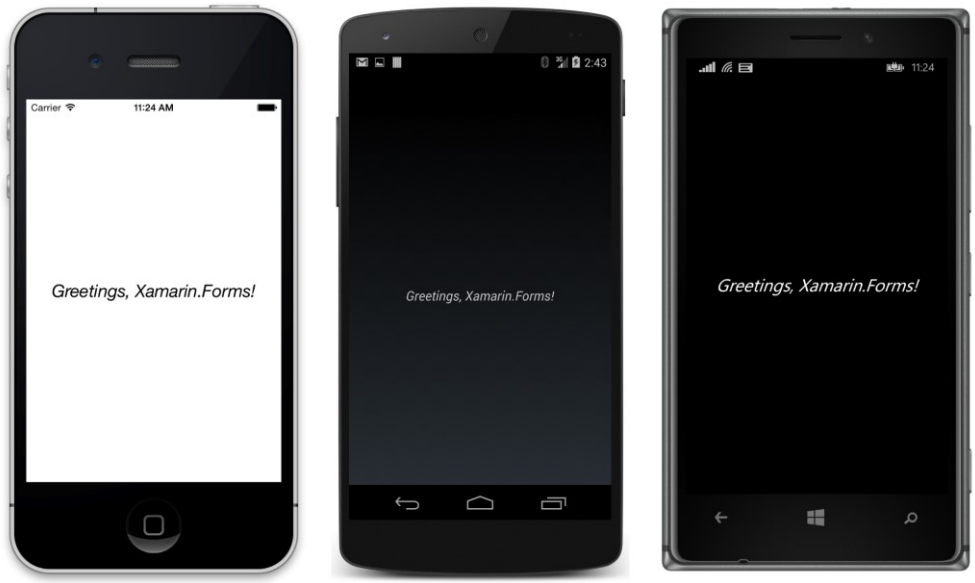
Another hard way is to specify a numeric font size rather than a member of the `NamedSize` enumeration. You'll need to get a feel for the sizes in the Xamarin.Forms platforms to choose a font size intelligently, and a crucial part of that information won't be discussed until Chapter 5.

The `SystemFontOfSize` method can also accept a member of the `FontAttributes` enumeration: `Bold`, `Italic`, or `None`.

You can try out the `Font` class in the **Greetings** program:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            Text = "Greetings, Xamarin.Forms!",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Font = Font.SystemFontOfSize(NamedSize.Large, FontAttributes.Italic)
        };
    }
}
```

Here it is on the three platforms:



`Font` also includes `WithSize` and `WithAttributes` instance methods that allow you to create a new `Font` value based on an existing `Font` value.

Formatted text

As you've seen, `Label` has a `Text` property that you can set to a string. But `Label` also has an alternative `FormattedText` property that constructs a paragraph with non-uniform formatting.

The `FormattedText` property is of type `FormattedString`, which has a `Spans` property of type `IList<Span>`, a collection of `Span` objects. Each `Span` object is a uniformly formatted chunk of text that is governed by four properties:

- `Text`
- `Font`
- `ForegroundColor`
- `BackgroundColor`

You can use the **Greetings** program to experiment with this. Here's one way to instantiate a `FormattedString` object and then add `Span` instances to its `Spans` collection property:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        FormattedString formattedString = new FormattedString();
```

```
        formattedString.Spans.Add(new Span
            {
                Text = "I "
            });
        formattedString.Spans.Add(new Span
            {
                Text = "love",
                Font = Font.SystemFontOfSize(NamedSize.Large, FontAttributes.Bold)
            });
        formattedString.Spans.Add(new Span
            {
                Text = " Xamarin.Forms!"
            });

        this.Content = new Label
        {
            FormattedText = formattedString,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Font = Font.SystemFontOfSize(NamedSize.Large)
        };
    }
}
```

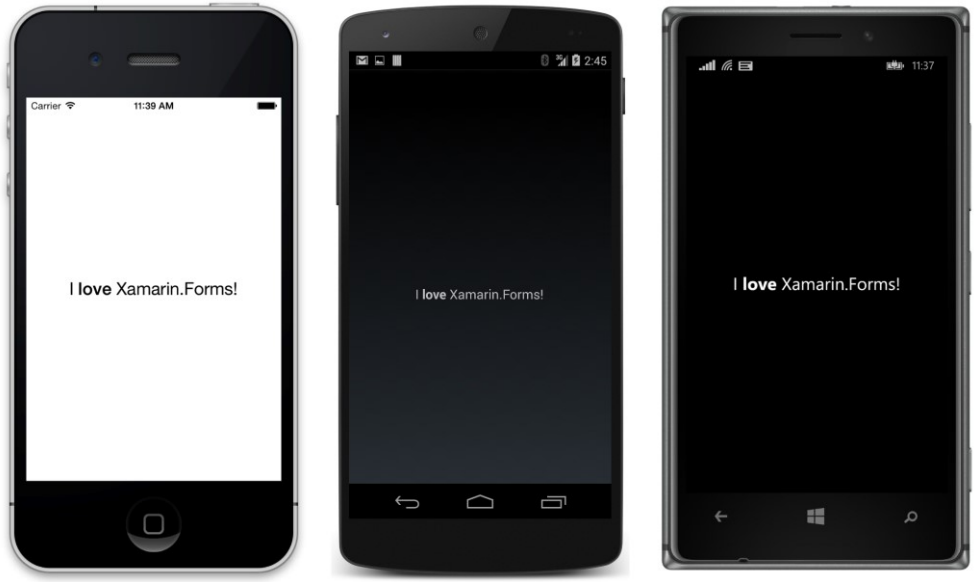
Each `Span` is created and passed to the `Add` method of the `Spans` collection.

However, it's possible to initialize the contents of the `Spans` collection by following it with a pair curly braces. Within these curly braces the `Span` objects are instantiated. Because no method calls are required, the entire `FormattedString` initialization can occur within the `Label` initialization:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Label
        {
            FormattedText = new FormattedString
            {
                Spans =
                {
                    new Span
                    {
                        Text = "I "
                    },
                    new Span
                    {
                        Text = "love",
                        Font = Font.SystemFontOfSize(NamedSize.Large, FontAttributes.Bold)
                    },
                    new Span
                    {
                        Text = " Xamarin.Forms!"
                    }
                }
```

```
            },

            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Font = Font.SystemFontOfSize(NamedSize.Large)
        };
    }
}
```

Regardless which approach you use, here's what it looks like:



Notice that `NamedSize.Large` is referenced twice in the code. That's not quite optimal. What you really want is for the word "love" to appear in bold with the same font size as the surrounding text. It makes more sense to define a `Font` value first that's used throughout the text, and then add a `Bold` attribute for the one word by using the `WithAttributes` method:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        Font baseFont = Font.SystemFontOfSize(NamedSize.Large);

        this.Content = new Label
        {
            FormattedText = new FormattedString
            {
                Spans =
                {
                    new Span
                    {
                        Text = "I "
```

```
                    },
                    new Span
                    {
                        Text = "love",
                        Font = baseFont.WithAttributes(FontAttributes.Bold)
                    },
                    new Span
                    {
                        Text = " Xamarin.Forms!"
                    }
                }
            },

            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Font = baseFont
        };
    }
}
```

Stacks of views

If you're like most programmers, as soon as you saw the list of static `Color` properties, you wanted to write a program to display them all, perhaps using the `Text` property of `Label` to identify the color and the `TextColor` property to show the actual color.

Although you could do this with a single `Label` using a `FormattedString` object, it's easier with multiple `Label` objects. Because multiple `Label` objects are involved, this job also requires some way to display all the `Label` objects in some kind of a list.

The `ContentPage` class defines a `Content` property of type `View` that you can set to an object—but only one object. Displaying multiple views requires a class that can have multiple children of type `View`. Such a class is `Layout<T>`, where `T` is of type `View`. The `Layout<T>` class defines a `Children` property of type `IList<T>`.

The `Layout<T>` class is abstract, but four classes derive from `Layout<T>`. They are:

- `AbsoluteLayout`
- `RelativeLayout`
- `StackLayout`
- `Grid`

Each of them arranges its view children in a characteristic manner.

The `StackLayout` arranges its children in a stack, and this seems ideal for the job of listing colors. You can use the `Add` method for adding children to the `Children` collection of a `StackLayout`

instance, or you can initialize the `Children` method with a collection of views, similar to the way the `Spans` collection of a `FormattedString` object was initialized earlier.

The `ColorList` program sets the `Content` property of the page to a `StackLayout` object, which then has its `Children` property initialized with 16 `Label` views:

```
class ColorListPage : ContentPage
{
    public ColorListPage()
    {
        this.Padding =
            new Thickness (5, Device.OnPlatform (20, 5, 5), 5, 5);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "White",
                    TextColor = Color.White
                },
                new Label
                {
                    Text = "Silver",
                    TextColor = Color.Silver
                },

                …

                new Label
                {
                    Text = "Purple",
                    TextColor = Color.Purple
                }
            }
        };
    }
}
```

You don't need to see the code for all 16 child `Label` views to get the idea. Here's the result:

Obviously, this isn't optimum. In each platform, one color isn't visible on all, and some of them are too faint to read well. On the iPhone, the list is dangerously close to overrunning the page, and it's not quite clear what would happen then.

`StackLayout` defines a `Spacing` property of type `double` that indicates how much space to leave between the children. By default, it's 6.0. You can set it to something smaller to help ensure that all the items will fit, for example, zero:

```
this.Content = new StackLayout
{
    Spacing = 0,
    Children =
    {
        new Label
        {
            Text = "White",
            TextColor = Color.White
        },

        …
```

Now all the `Label` views occupy only as much vertical space as required for the text. You can even set `Spacing` to negative values to make the items overlap!

You can set `Spacing` to something larger to see what happens when the items fall off the bottom of the page. What you'll discover is that the items do indeed seem to continue off the bottom of the screen and become inaccessible. Scrolling isn't automatic and must be added with a `ScrollView`. Moreover, the idea of explicitly creating 16 `Label` views in 16 blocks of very similar code is somewhat repulsive to programmers. Isn't there a way this can be automated?

Keep in mind that a Xamarin.Forms program is also a .NET program, and a .NET program can use .NET reflection to obtain information about all the classes and structures defined in an assembly, such as Xamarin.Forms.Core, including the members of those types. This suggests that obtaining the static fields and properties of the `Color` structure can be automated.

This is demonstrated by the **ReflectedColors** program. The ReflectedColorsPage.cs file requires a `using` directive for `System.Reflection`.

Most .NET reflection begins with a `Type` object. You can obtain a `Type` object for any class or structure using the C# `typeof` operator. For example, the expression `typeof(Color)` returns a `Type` object for the `Color` structure.

An extension method for the `Type` class named `GetTypeInfo` returns a `TypeInfo` object from which additional information can be obtained. But that's not required in this program. Instead, other extensions methods are defined for the `Type` class named `GetRuntimeFields` and `GetRuntime-Properties` that return the fields and properties of the type. These are in the form of collections of `FieldInfo` and `PropertyInfo` objects. From these, names of the properties can be obtained as well as values.

In two separate `foreach` statements the `ReflectedColorsPage` class loops through all the fields and properties of the `Color` structure. For all the public static members that return `Color` values, the two loops call `CreateColorLabel` to create a Label with the `Color` value and name, and then add that `Label` to the `StackLayout`.

By including all the public static fields and properties, the program lists `Color.Transparent`, `Color.Default`, and `Color.Accent` along with the 16 static fields displayed in the earlier program.

```
class ReflectedColorsPage : ContentPage
{
    public ReflectedColorsPage()
    {
        StackLayout stackLayout = new StackLayout();

        // Loop through the Color structure fields.
        foreach (FieldInfo fieldInfo in
                        typeof(Color).GetRuntimeFields())
        {
            if (fieldInfo.IsPublic &&
                fieldInfo.IsStatic &&
                fieldInfo.FieldType == typeof(Color))
            {
                stackLayout.Children.Add(
                    CreateColorLabel((Color) fieldInfo.GetValue(null),
                                     fieldInfo.Name));
            }
        }
```

```
        // Loop through the Color structure properties.
        foreach (PropertyInfo propInfo in
                        typeof(Color).GetRuntimeProperties())
        {
            MethodInfo methodInfo = propInfo.GetMethod;

            if (methodInfo.IsPublic &&
                methodInfo.IsStatic &&
                methodInfo.ReturnType == typeof(Color))
            {
                stackLayout.Children.Add(
                    CreateColorLabel((Color) propInfo.GetValue(null),
                                     propInfo.Name));
            }
        }

        this.Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);

        // Put the StackLayout in a ScrollView.
        this.Content = new ScrollView
        {
            Content = stackLayout
        };
    }

    Label CreateColorLabel(Color color, string name)
    {
        Color backgroundColor = Color.Default;

        if (color != Color.Default)
        {
            // Standard luminance calculation
            double luminance = 0.30 * color.R +
                               0.59 * color.G +
                               0.11 * color.B;

            backgroundColor = luminance > 0.5 ? Color.Black : Color.White;
        }

        // Create the Label.
        return new Label
        {
            Text = name,
            TextColor = color,
            BackgroundColor = backgroundColor,
            Font = Font.SystemFontOfSize(NamedSize.Large)
        };
    }
}
```

Towards the end of the constructor, the `StackLayout` is set to the `Content` property of a `ScrollView`, which is then set to the `Content` property of the page.

When adding children to a `StackLayout` in code, it's usually a good idea for the `StackLayout` to be disconnected from the page that will eventually display it. Every new child added to `StackLayout` causes the size of the `StackLayout` to change, and if the `StackLayout` were connected to the page, there would be a lot of layout activity going on that isn't really required.

The `CreateColorLabel` method in the class attempts to make each color visible by setting a contrasting background. The method calculates a luminance value based on a standard weighted average of the red, green, and blue components, and then selects a background of either white or black.

This technique won't work for `Transparent`, so that item can't be displayed at all, and the method treats `Color.Default` as a special case and displays that color (whatever it may be) against a `Color.Default` background.

Here are the results, which are still quite short of being aesthetically satisfying:

But you can scroll the display.

You'll recall that the `Layout<T>` class defines the `Children` property that `StackLayout` inherits. The generic `Layout<T>` class derives from the non-generic `Layout` class, and `ScrollView` also derives from this non-generic `Layout`. Theoretically, `ScrollView` is a type of layout object—even though it has only one child.

As you can see from the screenshot, the background color of the `Label` extends to the full width of the `StackLayout`, which means that each `Label` is as wide as the `StackLayout`.

Let's experiment a bit to get a better understanding of Xamarin.Forms layout. For these

experiments, you might want to temporarily give the StackLayout and the ScrollView distinct background colors:

```
class ReflectedColorsPage : ContentPage
{
    public ReflectedColorsPage()
    {
        StackLayout stackLayout = new StackLayout
        {
            BackgroundColor = Color.Blue
        };

        …

        this.Content = new ScrollView
        {
            BackgroundColor = Color.Red,
            Content = stackLayout
        };
    }
    …
}
```

Layout objects usually have transparent backgrounds by default, so although they occupy an area on the screen, they are not directly visible. Giving layout objects temporary colors is a great way to see exactly where they are on the screen. It's a good debugging technique for complex layouts.

You will discover that the blue StackLayout peaks out in the space between the individual Label views—this is a result of the default Spacing property of StackLayout—and also through the Label for Color.Default, which has a transparent background.

Try setting the HorizontalOptions property of all the Label views to LayoutOptions.Start:

```
        return new Label
        {
            Text = name,
            TextColor = color,
            BackgroundColor = backgroundColor,
            Font = Font.SystemFontOfSize(NamedSize.Large),
            HorizontalOptions = LayoutOptions.Start
        };
```

Now the blue background of the StackLayout is even more prominent because all the Label views occupy only as much horizontal space as the text require, and they are all pushed over to the left side. Because each Label view is a different width, this display looks even uglier than the first version!

Now remove the HorizontalOptions setting from the Label, and instead, set a HorizontalOptions on the StackLayout:

```
        StackLayout stackLayout = new StackLayout
        {
```

```
        BackgroundColor = Color.Blue,
        HorizontalOptions = LayoutOptions.Start
    };
```

Now the `StackLayout` becomes only as wide as the widest `Label`. The `StackLayout` hugs the labels within the `ScrollView`—at the left on iPhone and Android, and in the center (oddly enough) on Windows Phone—with the red background of the `ScrollView` now clearly in view.

As you begin constructing a tree of visual objects, these objects acquire a parent-child relationship. A parent object is sometimes referred to as the container of its child or children, because the child's location and size is contained within its parent.

By default, `HorizontalOptions` and `VerticalOptions` are set to `LayoutOptions.Fill`, which means that child views attempt to fill the parent container. (At least with the containers encountered so far. As you'll see, other layout classes have somewhat different behavior.) Even a `Label` fills its parent container by default, although without a background color the `Label` appears to occupy only as much space as it requires.

Setting a view's `HorizontalOptions` or `VerticalOptions` property to `LayoutOptions.Start`, `Center`, or `End` effectively forces the view to shrink down—either horizontally or vertically or both—to only the size the view requires.

A `StackLayout` has this same effect on its child's vertical size: Every child in a `StackLayout` occupies only as much height as it requires. Setting the `VerticalOptions` property on a child of a `StackLayout` to `Start`, `Center`, or `End` has no effect! However, the child views still expand to fill the width of the `StackLayout` except when the children are given a `HorizontalOptions` property other than `LayoutOptions.Fill`.

If a `StackLayout` is set to the `Content` property of a `ContentPage`, you can set `HorizontalOptions` or `VerticalOptions` on the `StackLayout`. These properties have two effects: First, they shrink the `StackLayout` width or height (or both) to the size of its children, and second, they govern where the `StackLayout` is positioned relative to the page.

If a `StackLayout` is in a `ScrollView`, then the `ScrollView` causes the `StackLayout` to be only as tall as the sum of the heights of its children. This is how the `ScrollView` can determine how to vertically scroll the `StackLayout`. You can continue to set `HorizontalOptions` on the `StackLayout` to control the width and horizontal placement.

However, what you do not want to do is set `VerticalOptions` on the `ScrollView` to `LayoutOptions.Start`, `Center`, or `End`. The `ScrollView` must be able to scroll its child content, and the only way `ScrollView` can do that is by forcing its child (usually a `StackLayout`) to assume a height reflecting only what the child needs, and then to use the height of its child and its own height to calculate how much to scroll that content.

If you set `VerticalOptions` on the `ScrollView` to `LayoutOption.Start`, `Center`, or `End`, you are effectively telling the `ScrollView` to be only as tall as it needs to be. But what is that? Because `ScrollView` can scroll its contents, it doesn't need to be any particular height, so it will

shrink down to nothing.

Although putting a `StackLayout` in a `ScrollView` is normal, putting a `ScrollView` in a `StackLayout` is dangerous. The `StackLayout` will force the `ScrollView` to have a height of only what it requires, and that required height is basically zero.

However, there is a way to put a `ScrollView` in a `StackLayout` successfully, and that will be demonstrated shortly.

The preceding discussion applies to a vertically oriented `StackLayout` and `ScrollView`. `StackLayout` has a property named `Orientation` that you can set to a member of the `StackOrientation` enumeration—`Vertical` (the default) or `Horizontal`. Similarly, `ScrollView` has a `ScrollOrientation` property that you set to a member of the `ScrollOrientation` enumeration. Try this:

```
public ReflectedColorsPage()
{
    StackLayout stackLayout = new StackLayout
    {
        BackgroundColor = Color.Blue,
        VerticalOptions = LayoutOptions.Center,
        Orientation = StackOrientation.Horizontal
    };

    …

    this.Content = new ScrollView
    {
        BackgroundColor = Color.Red,
        Content = stackLayout,
        Orientation = ScrollOrientation.Horizontal
    };
}
```

Now the `Label` views are stacked horizontally, the `StackLayout` has a height based on the height of its tallest child, and the `ScrollView` fills the page vertically but allows horizontal scrolling.

The layout expand option

You probably noticed that the `HorizontalOptions` and `VerticalOptions` properties are plurals, as if there's more than one option. These properties are generally set to a static field of the `LayoutOptions` structure—another plural.

The discussions so far have focused on the following static `LayoutOptions` fields:

- `LayoutOptions.Start`
- `LayoutOptions.Center`

- `LayoutOptions.End`
- `LayoutOptions.Fill`

The default—established by the `View` class—is `LayoutOptions.Fill`, which means that the view fills its container.

As you've seen, a `VerticalOptions` setting on a `Label` doesn't make a difference when the `Label` is a child of a vertical `StackLayout`. The `StackLayout` itself constrains the height of its children to only the height they require, so the child has no freedom to move vertically within that slot.

Be prepared for this rule to be slightly amended!

The `LayoutOptions` structure has four additional static fields not discussed yet:

- `LayoutOptions.StartAndExpand`
- `LayoutOptions.CenterAndExpand`
- `LayoutOptions.EndAndExpand`
- `LayoutOptions.FillAndExpand`

As you'll recall, `LayoutOptions` has two instance properties named `Alignment` and `Expands`. These four instances of `LayoutOptions` all have the `Expands` property set to `true`. This `Expands` property can be very useful for managing the layout of the page, but it can be confusing on first encounter. Here are the requirements for `Expands` to play a role in a vertical `StackLayout`:

- The vertical `StackLayout` must have a height that is less than the height of its container. In other words, some extra unused vertical space must exist in the `StackLayout`.
- That first requirement implies that the vertical `StackLayout` cannot have a `VerticalOptions` setting of `Start`, `Center`, or `End` because that would cause the `StackLayout` to have a height equal to the height of its children and it would have no extra space.
- At least one child of the `StackLayout` must have a `VerticalOptions` setting with the `Expands` property set to `true`.

If these conditions are satisfied, the `StackLayout` allocates the extra vertical space equally among all the children that have a `VerticalOptions` setting with `Expands` equal to `true`. How the child occupies that space depends on the `Alignment` setting: `Start`, `Center`, `End`, or `Fill`.

Here's a program named **VerticalOptionsDemo** that uses reflection to create `Label` objects with all the possible `VerticalOptions` settings in a vertical `StackLayout`. The background and foreground colors are alternated so that you can see exactly how much space each `Label` occupies. The program uses Language Integrated Query (LINQ) to sort the fields of the `LayoutOptions` structure in a visually more illuminating manner:

```
class VerticalOptionsDemoPage : ContentPage
{
```

```
    public VerticalOptionsDemoPage()
    {
        Color[] colors = { Color.Yellow, Color.Blue };
        int flipFlopper = 0;

        // Create Labels sorted by LayoutAlignment property.
        IEnumerable<Label> labels =
            from field in typeof(LayoutOptions).GetRuntimeFields()
            where field.IsPublic && field.IsStatic
            orderby ((LayoutOptions)field.GetValue(null)).Alignment
            select new Label
            {
                Text = "VerticalOptions = " + field.Name,
                VerticalOptions = (LayoutOptions)field.GetValue(null),
                XAlign = TextAlignment.Center,
                Font = Font.SystemFontOfSize(NamedSize.Large),
                TextColor = colors[flipFlopper],
                BackgroundColor = colors[flipFlopper = 1 - flipFlopper]
            };

        // Transfer to StackLayout.
        StackLayout stackLayout = new StackLayout();

        foreach (Label label in labels)
        {
            stackLayout.Children.Add(label);
        }

        this.Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
        this.Content = stackLayout;
    }
}
```

You might want to study the results a little:

The `Label` views with yellow text on blue backgrounds are those `VerticalOptions` properties are set to `LayoutOptions` values without the `Expands` flag set. If the `Expands` flag is not set on the `LayoutOptions` value of an item in a vertical `StackLayout`, the `VerticalOptions` setting is ignored. As you can see, the `Label` occupies only as much vertical space as it needs in the vertical `StackLayout`.

The total height of the children in this `StackLayout` is less than the height of the `StackLayout`, so the `StackLayout` has extra space. It contains four children with `VerticalOptions` properties set to `LayoutOptions` values with the `Expands` flag set, so this extra space is allocated equally among those four children.

In these four cases—the `Label` views with blue text on yellow backgrounds—the `Alignment` property of the `LayoutOptions` value indicates how the child is aligned within the area that includes the extra space. The first one—with the `VerticalOptions` property set to `LayoutOptions.StartAndExpand`—is above this extra space. The second (`CenterAndExpand`) is in the middle of the extra space. The third (`EndAndExpand`) is below the extra space. However, in all these three cases, the `Label` is only getting as much vertical space as it needs, as indicated by the background color. The rest of the space belongs to the `StackLayout`, which shows the background color of the page.

The last `Label` has its `VerticalOptions` property set to `LayoutOptions.FillAndExpand`. In this case, the `Label` occupies the entire area including the extra space, as the large area of yellow background indicates. The text is at the top of this area; that's because the default setting of `YAlign` is `TextAlignment.Start`. Set it to something else to position the text vertically within the area.

The `Expands` property of `LayoutOptions` plays a role only when the view is a child of a `StackLayout`. In other contexts, it's superfluous.

Frame and BoxView

Two simple rectangular views are often useful for presentation purposes:

The `BoxView` is a simple filled rectangle. It derives from `View` and defines a `Color` property that's transparent by default.

The `Frame` displays a rectangular border surrounding some content. `Frame` derives from `Layout` by way of `ContentView`, from which it inherits a `Content` property. The content of a `Frame` can be a single view or a layout containing a bunch of views. From `VisualElement`, `Frame` inherits a `BackgroundColor` property that's white on the iPhone but transparent on Android and Windows Phone. From `Layout`, `Frame` inherits a `Padding` property that it initializes to 20 units on all sides to give the content a little breathing room. `Frame` itself defines an `OutlineColor` property that is transparent by default, and a `HasShadow` property that is `true` by default, but the shadow only shows up on the iPhone.

If the `BoxView` or `Frame` is not constrained in size in any way—that is, if it's not in a `StackLayout` and has its `HorizontalOptions` and `VerticalOptions` set to default values of `LayoutOptions.Fill`—these views expand to fill their containers.

For example, try modifying the **Greetings** program so that the `GreetingsPage` constructor puts the `Label` in a `Frame`. Be sure to give the `Frame` an `OutlineColor`:

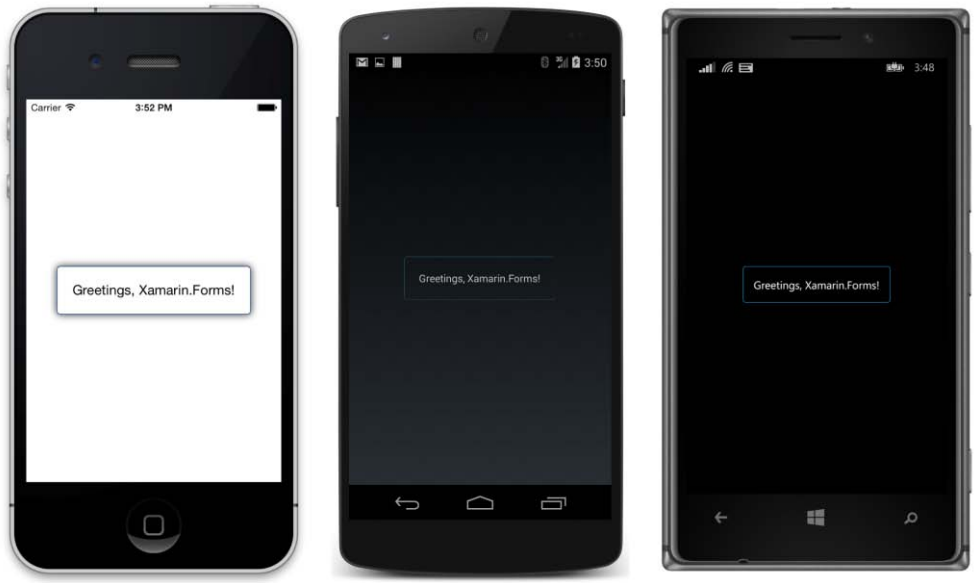
```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Frame
        {
            OutlineColor = Color.Accent,
            Content = new Label
            {
                Text = "Greetings, Xamarin.Forms!"
            }
        };
    }
}
```

It's likely you won't even notice the `Frame` because it's expanded to fill the page. Set the `HorizontalOptions` and `VerticalOptions` properties on the `Frame` to `LayoutOptions.Center` (for example):

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new Frame
        {
            OutlineColor = Color.Accent,
```

```
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,

            Content = new Label
            {
                Text = "Greetings, Xamarin.Forms!"
            }
        };
    }
}
```

Now the `Frame` hugs the text (but with a 20-unit default padding) in the center of the page:



The `BoxView` is transparent by default, so you'll need to set a color:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new BoxView
        {
            Color = Color.Accent
        }
    ]
}
```

The `BoxView` fills the whole area of its container, just as `Label` does with its default `HorizontalOptions` or `VerticalOptions` settings. Try setting those properties on the `BoxView`:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
```

```
    {
        this.Content = new BoxView
        {
            Color = Color.Accent,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        }
    }
}
```

Now the `BoxView` will assume its default dimensions, which are 40 units square. The `BoxView` is that size because the `BoxView` initializes its `WidthRequest` and `HeightRequest` properties to 40.

These `WidthRequest` and `HeightRequest` properties require a little explanation:

`VisualElement` defines `Width` and `Height` properties, but these properties are read-only. `VisualElement` also defines `WidthRequest` and `HeightRequest` properties that are both settable and gettable. Normally, all these properties are initialized to –1 (which effectively means they are undefined), but some `View` derivatives, such as `BoxView`, set the `WidthRequest` and `HeightRequest` properties to specific values.

Following the layout of a page, the `Width` and `Height` properties indicate actual dimensions of the view—the area that the view occupies on the screen. Because `Width` and `Height` are read-only, they are for informational purposes only.

If you want a `View` to be a specific size you can set the `WidthRequest` and `HeightRequest` properties. But these properties indicate (as the named suggests) a *requested* size, or a *preferred* size. If the view is allowed to fill its container, these properties will be ignored. But if not, these properties will govern the view's size.

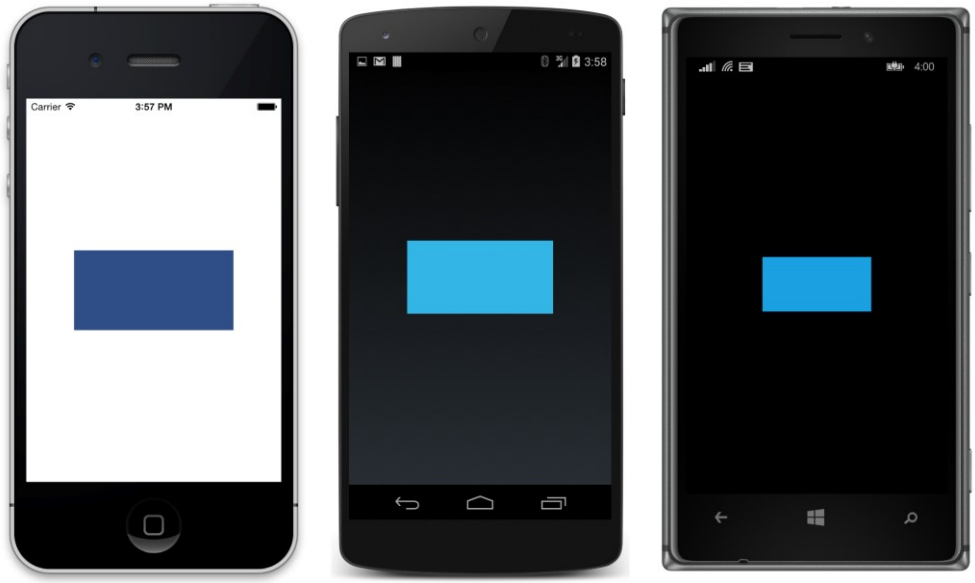
`BoxView` sets its own `WidthRequest` and `HeightRequest` properties to 40. You can think of these settings as a size that `BoxView` would like to be if nobody else has any opinions in the matter. You've already seen that `WidthRequest` and `HeightRequest` are ignored when the `BoxView` is allowed to fill the page. The `WidthRequest` kicks in if the `HorizontalOptions` is set to `LayoutOptions.Left`, `Center`, or `Right`, or if the `BoxView` is a child of a horizontal `StackLayout`. The `HeightRequest` behaves similarly.

Try this:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        this.Content = new BoxView
        {
            Color = Color.Accent,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            WidthRequest = 200,
            HeightRequest = 100
```

```
    }
}
```

Now we get a `BoxView` with that specific size:



Let's use both `Frame` and `BoxView` in an enhanced color list. The **ColorBlock** program has a page constructor that is virtually identical to the one in **ReflectedColors**, except that it calls a method named `CreateColorView` rather than `CreateColorLabel`. Here's that method:

```
class ColorBlockPage : ContentPage
{
    …

    View CreateColorView(Color color, string name)
    {
        return new Frame
        {
            Content = new StackLayout
            {
                Orientation = StackOrientation.Horizontal,
                Spacing = 15,
                Children =
                {
                    new BoxView
                    {
                        Color = color
                    },
                    new Label
                    {
                        Text = name,
                        Font = Font.BoldSystemFontOfSize(NamedSize.Medium),
```

```
                    VerticalOptions = LayoutOptions.Center
                },
                new Label
                {
                    Text = String.Format("{0:X2}-{1:X2}-{2:X2}",
                                         (int)(255 * color.R),
                                         (int)(255 * color.G),
                                         (int)(255 * color.B)),
                    VerticalOptions = LayoutOptions.Center,
                    IsVisible = color != Color.Default
                }
            }
        }
    };
  }
}
```

The `CreateColorView` method returns a `Frame` containing a horizontal `StackLayout` with a `BoxView` indicating the color, and two `Label` views for the name of the color and its RGB composition. This RGB display is meaningless for the `Color.Default` value, so the second `Label` has its `IsVisible` property set to `false` in that case. The `Label` still exists, but it's treated as non-existent when the page is rendered.

Now this is a scrollable color list that's beginning to be something we can take a little pride in:

A ScrollView in a StackLayout?

It is common to put a `StackLayout` in a `ScrollView`, but can you put a `ScrollView` in a `StackLayout`? And why would you even want to?

It's a general rule in layout systems like the one in Xamarin.Forms that you can't put a scroll in a stack. A `ScrollView` needs to have a specific height in order to compute the difference between the height of its content and its own height. That difference is the amount that the `ScrollView` can scroll its contents. If the `ScrollView` is in a `StackLayout`, it doesn't get that specific height. The `StackLayout` wants the `ScrollView` to be as short as possible, and that's either the height of the `ScrollView` contents, or zero, and neither solution works.

So why would you want a `ScrollView` in a `StackLayout` anyway?

Sometimes it's precisely what you need. Consider a primitive e-book reader that implements scrolling. You might want a `Label` at the top of the page always displaying the book's title, followed by a `ScrollView` containing a `StackLayout` with the content of the book itself. It would be convenient for that `Label` and the `ScrollView` to be children of a `StackLayout` that fills the page.

With Xamarin.Forms, such a thing is possible. If you give the `ScrollView` a `VerticalOptions` setting of `LayoutOptions.FillAndExpand`, it can indeed be a child of a `StackLayout`. The `StackLayout` will give the `ScrollView` all the extra space not required by the other children, and the `ScrollView` will then have a specific height.

The **BlackCat** project displays the text of Edgar Allan Poe's short story "The Black Cat," which is stored in a text file named TheBlackCat.txt in a one-line-per-paragraph format.

It is sometimes convenient to embed files that an application requires (such as this short story) right in the program executable or—in the case of a Xamarin.Forms application—right in the Portable Class Library DLL. These files are known as embedded resources, and that's what TheBlackCat.txt file is in this program.

BlackCat is a PCL application. It is possible to do something similar in an SAP application, but it's a little more difficult because the three platforms reference the resource with different names.

To make an embedded resource in either Visual Studio or Xamarin Studio, you'll probably first want to create a folder in the project by selecting the **Add > New Folder** option from the project menu. A folder for text files might be called **Texts**, for example. The folder is optional, but it helps organize program assets. Then, to that folder, you can select the **Add > Existing Item** in Visual Studio, or **Add > Add Files** in Xamarin Studio. Navigate to the file, select and click **Add** in Visual Studio, or **Open** in Xamarin Studio.

Now here's the important part: Once the file is part of the project, bring up the **Properties** dialog from the menu associated with the file. Specify that the **Build Action** for the file is **Embedded-Resource**. This is an easy step to forget, but it is essential.

This was done for the **BlackCat** project, and consequently the TheBlackCat.txt file is embedded in the BlackCat.dll file.

In code, the file can be retrieved by calling the `GetManifestResourceStream` method defined by the `Assembly` class in the `System.Reflection` namespace. To get the assembly of the PCL, all you need to do is get the `Type` of any class defined in the assembly. You can use `typeof` with the page type you've derived from `ContentPage`, or `GetType` on the instance of that class. Then call `GetTypeInfo` on this `Type` object. `Assembly` is a property of the resultant `TypeInfo` object:

```
Assembly assembly = this.GetType().GetTypeInfo().Assembly;
```

In the `GetManifestResourceStream` method of `Assembly` you'll need to specify the name of the resource. For embedded resources, that name is not the filename of the resource but the resource ID. It's easy to confuse these because that ID might look vaguely like a fully-qualified filename.

The resource ID begins with the default namespace of the assembly. This is not the .NET namespace! To get the default namespace of the assembly in Visual Studio, select **Properties** from the project menu, and in the properties dialog, select **Library** at the left and look for the **Default namespace** field. In Xamarin Studio, select **Options** from the project menu, and in the **Project Options** dialog, select **Main Settings** at the left, and look for a field labeled **Default Namespace**.

For the **BlackCat** project, that default namespace is the same as the assembly: "BlackCat". However, you can actually set that default namespace to whatever you want.

The resource ID begins with that default namespace, followed by a period, followed by the folder name you might have used, followed by another period and the filename. For this example, the resource ID is "BlackCat.Texts.TheBlackCat.txt" and that's what you'll see passed to the `GetManifest-ResourceStream` method in the code. The method returns a .NET `Stream` object, and from that a `StreamReader` can be created to read the lines of text.

It's a good idea to use `using` statements with the `Stream` object returned from `GetManifest-ResourceStream` and the `StreamReader` object because that will properly dispose of the objects when they're no longer needed.

For layout purposes, the `BlackCatPage` constructor creates two `StackLayout` objects: `mainStack` and `textStack`. The first line from the file (containing the story's title and author) becomes a bolded and centered `Label` in `mainStack`; all the subsequent lines go in `textStack`. The `mainStack` also contains a `ScrollView` with `textStack`.

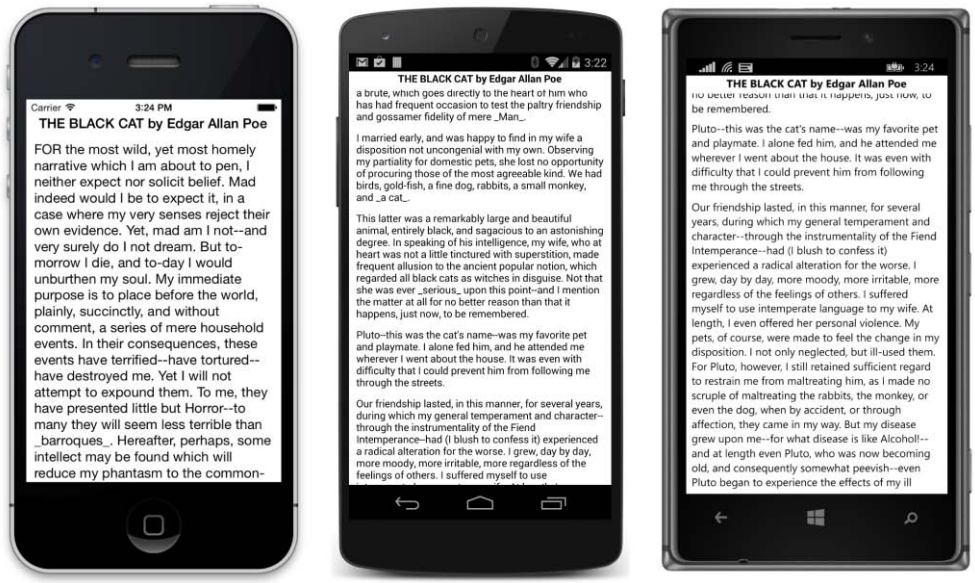
```
class BlackCatPage : ContentPage
{
    public BlackCatPage()
    {
        StackLayout mainStack = new StackLayout();
        StackLayout textStack = new StackLayout
        {
            Padding = new Thickness(5),
            Spacing = 10
```

```
};

// Get access to the text resource.
Assembly assembly = this.GetType().GetTypeInfo().Assembly;
string resource = "BlackCat.Texts.TheBlackCat.txt";

using (Stream stream = assembly.GetManifestResourceStream (resource))
{
    using (StreamReader reader = new StreamReader (stream))
    {
        bool gotTitle = false;
        string line;

        // Read in a line (which is actually a paragraph).
        while (null != (line = reader.ReadLine()))
        {
            Label label = new Label
            {
                Text = line,

                // Black text for ebooks!
                TextColor = Color.Black
            };

            if (!gotTitle)
            {
                // Add first label (the title to mainStack.
                label.HorizontalOptions = LayoutOptions.Center;
                label.Font = Font.SystemFontOfSize(NamedSize.Medium,
                                                   FontAttributes.Bold);
                mainStack.Children.Add(label);
                gotTitle = true;
            }
            else
            {
                // Add subsequent labels to textStack.
                textStack.Children.Add(label);
            }
        }
    }
}

// Put the textStack in a ScrollView with FillAndExpand.
ScrollView scrollView = new ScrollView
{
    Content = textStack,
    VerticalOptions = LayoutOptions.FillAndExpand,
    Padding = new Thickness(5, 0)
};

// Add the ScrollView as a second child of mainStack.
mainStack.Children.Add(scrollView);

// Set page content to mainStack.
```

```
            this.Content = mainStack;

            // White background for ebooks!
            this.BackgroundColor = Color.White;

            // Add some iOS padding for the page
            this.Padding = new Thickness (0, Device.OnPlatform (20, 0, 0), 0, 0);
        }
}
```

Notice that the `ScrollView` has its `VerticalOptions` property set to `LayoutOptions.FillAndExpand`. Without that, this program won't work. With it, the text is scrollable while the title stays in place.

Because this is basically an e-book reader, and humans have been reading black text on white paper for hundreds of years, the `BackgroundColor` of the page is set to white and the `TextColor` of each `Label` is set to black:



It is also possible to write this program using a Shared Asset Project rather than a PCL. However, if you put an embedded resource into an SAP, the folder name is not part of the resource ID. It's basically ignored. (This treatment of the folder name might change in the future.) Also, because the resource actually becomes part of the application project, you'll need the default namespace for the application, and that's different for each platform. The code to set the resource variable might look like this:

```
#if __IOS__
            string resource = "BlackCat.iOS.TheBlackCat.txt";
#elif WINDOWS_PHONE
            string resource = "BlackCat.WinPhone.TheBlackCat.txt";
#else
```

```
            string resource = "BlackCat.Droid.TheBlackCat.txt";
#endif
```

Depending on how the project was created, the default namespace for the Android project might be "projectname.Android" or "projectname.Droid". It's easy to check which it is.

If you're having problems referencing an embedded resource, you might be using an incorrect name. Try calling `GetManifestResourceNames` on the `Assembly` object to get a list of the resource IDs of all embedded resources.

The button for commands

While the `Label` is the most basic of presentation views, the `Button` is probably the most basic of interactive views. The `Button` signals a command. It's the user's way of telling the program to initiate some action—to do something.

Visually, a Xamarin.Forms button is some text surrounded by an optional border. When a finger presses on the button and is then released while still within the button boundaries, the button fires a `Clicked` event.

Event handlers in Xamarin.Forms are like typical .NET event handlers in that they have two arguments: The first argument is the object firing the event; for the `Clicked` event defined by the `Button`, the first argument is the particular `Button` that's been tapped. The second argument to the event handler sometimes provides more information about the event; in the case of the `Clicked` event, the second argument is simply an `EventArgs` object that provides no additional information.

Here's a program with a `Button` sharing a `StackLayout` with a `ScrollView` containing another `StackLayout`. Every time the `Button` is clicked, the program adds a new `Label` to the scrollable `StackLayout`, in effect logging all the button clicks:

```
class ButtonLoggerPage : ContentPage
{
    StackLayout loggerLayout = new StackLayout();

    public ButtonLoggerPage()
    {
        // Create the Button and attach Clicked handler.
        Button button = new Button
        {
            Text = "Log the Click Time"
        };
        button.Clicked += OnButtonClicked;

        this.Padding = new Thickness(5, Device.OnPlatform(20, 0, 0), 5, 0);

        // Assemble the page.
        this.Content = new StackLayout
        {
```

```
            Children =
            {
                button,
                new ScrollView
                {
                    VerticalOptions = LayoutOptions.FillAndExpand,
                    Content = loggerLayout
                }
            }
        };
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        // Add Label to scrollable StackLayout.
        loggerLayout.Children.Add(new Label
        {
            Text = "Button clicked at " + DateTime.Now.ToString("T")
        });
    }
}
```

In the programs in this book, event handlers are given names beginning with the word `On`, followed by some kind of identification of the view firing the event (sometimes just the view type), followed by the event name. The resultant name in this case is `OnButtonClicked`.

The `Clicked` handler is attached to the `Button` right after the `Button` is created:

```
button.Clicked += OnButtonClicked;
```

Then the page is assembled. Notice that the `ScrollView` has its `VerticalOptions` set to `FillAndExpand` so that it can share the `StackLayout` with the `Button` and still be visible and scrollable.

Here's the display after a few clicks:

As you can see, the `Button` looks a little different in the three environments, but by default it always fills the area available for it and centers the text inside.

`Button` defines several properties that let you customize its appearance:

- `Font`
- `TextColor`
- `BorderColor`
- `BorderWidth`
- `BorderRadius`

`Button` also inherits the `BackgroundColor` property from `VisualElement`.

Some properties might not work on all platforms. On the iPhone you need to set `BorderWidth` to a positive value for a border to be displayed, but that's normal for an iPhone button. The Android button won't display a border unless the `BackgroundColor` is set, and then it requires a non-default setting of `BorderColor` and a positive `BorderWidth`. The `BorderRadius` property, which is intended to round off the sharp corners of the border, doesn't work on Windows Phone.

Suppose you wrote a program similar to this, but you did not save the `loggerLayout` object as a field. Could you get access to that `StackLayout` object in the `Clicked` event handler?

Yes! It's possible to obtain parent and child visual elements by using a technique sometimes called "walking the tree." All the visual elements of the page comprise a visual tree, and you can walk that tree (or perhaps climb the tree?) using properties of the visual elements.

The `sender` argument to the `OnButtonClicked` handler is the object firing the event, in this case the `Button`, so begin the `Clicked` handler by casting that argument:

```
Button button = (Button)sender;
```

The `Button` is a child of a `StackLayout`, so that `StackLayout` is accessible from the `ParentView` property. Again, some casting is required:

```
StackLayout outerLayout = (StackLayout)button.ParentView;
```

The second child of this `StackLayout` is the `ScrollView`, so the `Children` property can be indexed to obtain that:

```
ScrollView scrollView = (ScrollView)outerLayout.Children[1];
```

The `Content` property of this `ScrollView` is exactly the `StackLayout` we were looking for:

```
StackLayout loggerLayout = (StackLayout)scrollView.Content;
```

Of course, the danger in doing something like this is that you might change the layout some day and forget to change your tree-walking code as well.

Sharing button clicks

If a program contains multiple `Button` views, each `Button` can have its own `Clicked` handler. However, in some cases it might be more convenient for multiple `Button` views to share a common `Clicked` handler.

Consider a calculator program. Each of the buttons labeled '0' through '9' basically does the same thing, and having ten separate `Clicked` handlers for these ten buttons—even if they shared some common code—simply wouldn't make much sense.

You've seen how the first argument to the `Clicked` handler can be cast to an object of type `Button`. But how do you know which `Button` it is?

One approach is to store all the `Button` objects as fields and then compare the `Button` object firing the event with these fields.

The **TwoButtons** program demonstrates this technique. This program is similar to the previous program but with two buttons—one to add `Label` objects to the `StackLayout`, and the other to remove them. The two `Button` objects are stored as fields so that the `Clicked` handler can determine which one fired the event:

```
class TwoButtonsPage : ContentPage
{
    Button addButton, removeButton;
    StackLayout loggerLayout = new StackLayout();

    public TwoButtonsPage()
    {
        // Create the Button views and attach Clicked handlers.
```

```
        addButton = new Button
        {
            Text = "Add",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        addButton.Clicked += OnButtonClicked;

        removeButton = new Button
        {
            Text = "Remove",
            HorizontalOptions = LayoutOptions.CenterAndExpand,
            IsEnabled = false
        };
        removeButton.Clicked += OnButtonClicked;

        this.Padding = new Thickness(5, Device.OnPlatform(20, 0, 0), 5, 0);

        // Assemble the page.
        this.Content = new StackLayout
        {
            Children =
            {
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        addButton,
                        removeButton
                    }
                },

                new ScrollView
                {
                    VerticalOptions = LayoutOptions.FillAndExpand,
                    Content = loggerLayout
                }
            }
        };
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        if (button == addButton)
        {
            // Add Label to scrollable StackLayout.
            loggerLayout.Children.Add(new Label
                {
                    Text = "Button clicked at " + DateTime.Now.ToString("T")
                });
        }
        else
```

```
        {
            // Remove topmost Label from StackLayout
            loggerLayout.Children.RemoveAt(0);
        }

        // Enable "Remove" button only if children are present.
        removeButton.IsEnabled = loggerLayout.Children.Count > 0;
    }
}
```

The two buttons are given `HorizontalOptions` values of `CenterAndExpand` so that they can be nicely arranged on the screen using a horizontal `StackLayout`.

Notice that when the `Clicked` handler detects `removeButton`, it simply calls the `RemoveAt` method on the `Children` property. What happens if there are no children?

It can't happen in this program! When the application begins, the `IsEnabled` property of the `removeButton` is initialized to `false`. When a button is disabled in this way, it appears to be non-functional, and it actually is non-functional. It does not fire `Clicked` events. Towards the end of the `Clicked` handler, the `IsEnabled` property on `removeButton` is set to `true` only if the `loggerLayout` has at least one child.

This illustrates a good general rule: If your code needs to determine if a button `Clicked` event is valid, it's likely much better to prevent invalid button clicks entirely by setting `IsEnabled` to `false`.

Anonymous event handlers

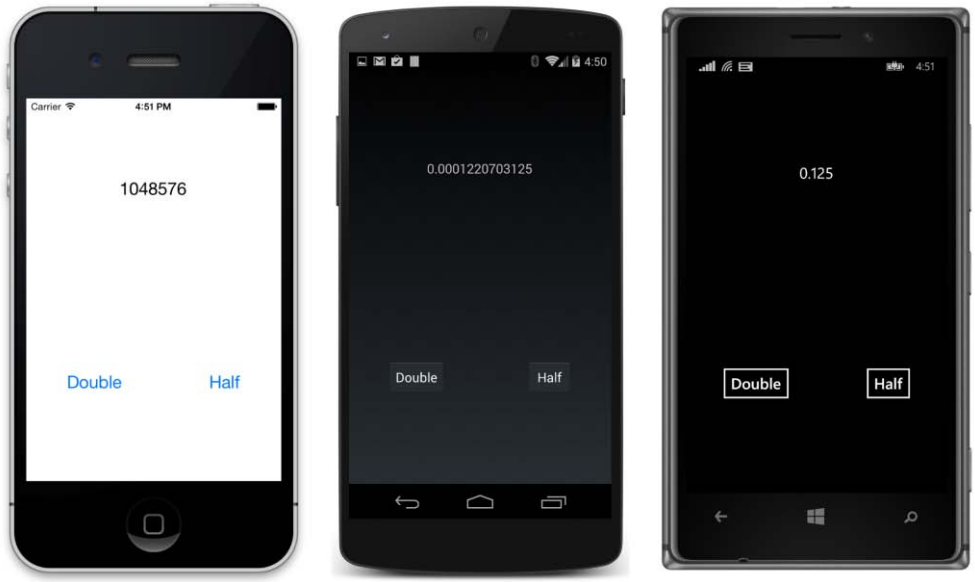
Many C# programmers these days like to define small event handlers as anonymous lambda functions. This allows the event handling code to be very close to the instantiation and initialization of the object firing the event instead of somewhere else in the file. It also allows referencing objects within the event handler without storing those objects as fields.

Here's a program named **ButtonLambdas** that has a `Label` displaying a number and two buttons. One button doubles the number, and the other halves the number. Normally the number and label variables would be saved as fields. But because the anonymous event handlers are defined right in the constructor after these variables are defined, the event handlers have access to them:

```
class ButtonLambdasPage : ContentPage
{
    public ButtonLambdasPage()
    {
        // Number to manipulate.
        double number = 1;

        // Create the Label for display.
        Label label = new Label
        {
            Text = number.ToString(),
            Font = Font.SystemFontOfSize(NamedSize.Large),
            HorizontalOptions = LayoutOptions.Center,
```

```
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        // Create the first Button and attach Clicked handler.
        Button timesButton = new Button
        {
            Text = "Double",
            Font = Font.SystemFontOfSize(NamedSize.Large),
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        timesButton.Clicked += (sender, args) =>
            {
                number *= 2;
                label.Text = number.ToString();
            };

        // Create the second Button and attach Clicked handler.
        Button divideButton = new Button
        {
            Text = "Half",
            Font = Font.SystemFontOfSize(NamedSize.Large),
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        divideButton.Clicked += (sender, args) =>
            {
                number /= 2;
                label.Text = number.ToString();
            };

        // Assemble the page.
        this.Content = new StackLayout
        {
            Children =
            {
                label,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    Children =
                    {
                        timesButton,
                        divideButton
                    }
                }
            }
        };
    }
}
```

Like the previous program, the two buttons share a horizontal `StackLayout`:



The disadvantage of defining event handlers as anonymous lambda functions is that they can't be shared among multiple views.

Distinguishing views with IDs

In the **TwoButtons** program you saw a technique for sharing an event handler that distinguishes views by comparing objects. This works fine when there aren't very many views to distinguish, but it would be a terrible approach for a calculator program.

The `Element` class defines a `StyleId` of type `string` specifically for the purpose of identifying views. Set it to whatever is convenient for the application. You can test the values using `if` and `else` statements, or in a `switch` and `case`, or you can use a `Parse` method to convert the strings into numbers or enumeration members.

The following program isn't a calculator, but it is a numeric keypad, which is certainly part of a calculator. The program is called **SimplestKeypad** and uses a `StackLayout` for organizing the rows and columns of keys. (One of the intents of this program is to demonstrate that `StackLayout` is not quite the right tool for this job!)

The program creates a total of four `StackLayout` instances. The `mainStack` is vertically oriented; three horizontal `StackLayout` objects arrange the nine non-zero digit buttons. To keep things simple, the keypad is arranged with telephone ordering rather than calculator ordering:

```
class SimplestKeypadPage : ContentPage
{
    Label displayLabel;
    Button backspaceButton;
```

```
public SimplestKeypadPage()
{
    // Create a vertical stack for the entire keypad.
    StackLayout mainStack = new StackLayout
    {
        VerticalOptions = LayoutOptions.Center,
        HorizontalOptions = LayoutOptions.Center
    };

    // First row is the Label.
    displayLabel = new Label
    {
        Font = Font.SystemFontOfSize(NamedSize.Large),
        VerticalOptions = LayoutOptions.Center,
        XAlign = TextAlignment.End
    };
    mainStack.Children.Add(displayLabel);

    // Second row is the backspace Button.
    backspaceButton = new Button
    {
        Text = "\u21E6",
        Font = Font.SystemFontOfSize(NamedSize.Large),
        IsEnabled = false
    };
    backspaceButton.Clicked += OnBackspaceButtonClicked;
    mainStack.Children.Add(backspaceButton);

    // Now do the 10 number keys.
    StackLayout rowStack = null;

    for (int num = 1; num <= 10; num++)
    {
        if ((num - 1) % 3 == 0)
        {
            rowStack = new StackLayout
            {
                Orientation = StackOrientation.Horizontal
            };
            mainStack.Children.Add(rowStack);
        }

        Button digitButton = new Button
        {
            Text = (num % 10).ToString(),
            Font = Font.SystemFontOfSize(NamedSize.Large),
            StyleId = (num % 10).ToString()
        };
        digitButton.Clicked += OnDigitButtonClicked;

        // For the zero button, expand to fill horizontally.
        if (num == 10)
        {
            digitButton.HorizontalOptions = LayoutOptions.FillAndExpand;
```

```
            }
            rowStack.Children.Add(digitButton);
        }

        this.Content = mainStack;
    }

    void OnDigitButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        displayLabel.Text += (string)button.StyleId;
        backspaceButton.IsEnabled = true;
    }

    void OnBackspaceButtonClicked(object sender, EventArgs args)
    {
        string text = displayLabel.Text;
        displayLabel.Text = text.Substring(0, text.Length - 1);
        backspaceButton.IsEnabled = displayLabel.Text.Length > 0;
    }
}
```

The ten number keys share a single `Clicked` handler. The `StyleId` property indicates the number associated with the key, so it can be appended to the string displayed by the `Label`. The `StyleId` happens to be identical to the `Text` property of the `Button`, and the `Text` property could be used instead, but in the general case, things aren't quite that convenient.

The backspace `Button` is sufficiently different in function to warrant its own `Clicked` handler, although it would surely be possible to combine the two methods into one to take advantage of any code they have in common.

To give the keypad a slightly larger size, all the text is given a `Font` object using `NamedSize.Large`. Here are the three renderings of the **KeypadStack** program:

Of course you'll want to press the keys maniacally to see how the program responds with a really large string of digits, and you'll discover that it doesn't adequately anticipate such a thing. When the `Label` gets too wide, it begins to govern the overall width of the vertical `StackLayout`, and the buttons start shifting as well.

But even before that, you might notice a little irregularity in the `Button` widths, particularly on the Windows Phone. The widths of the individual `Button` objects are based on their content, and in many fonts the widths of the decimal digits are not the same.

Can you fix this problem with the `Expands` flag on the `HorizontalOptions` property? No. The `Expands` flag causes extra space to be distributed equally among the views in the `StackLayout`. Each view will increase additively by the same amount, so they still won't be the same width. For example, take a look at the two buttons on the top of the **TwoButtons** or **ButtonLambdas** program. They have their `HorizontalOptions` properties set to `FillAndExpand`, but they are different widths because the width of the `Button` content is different.

A better solution for these programs is the layout known as the `Grid`, coming up in Chapter 5.

Meanwhile, let's declare ourselves sufficiently knowledgeable to begin building a real application.

CHAPTER 3

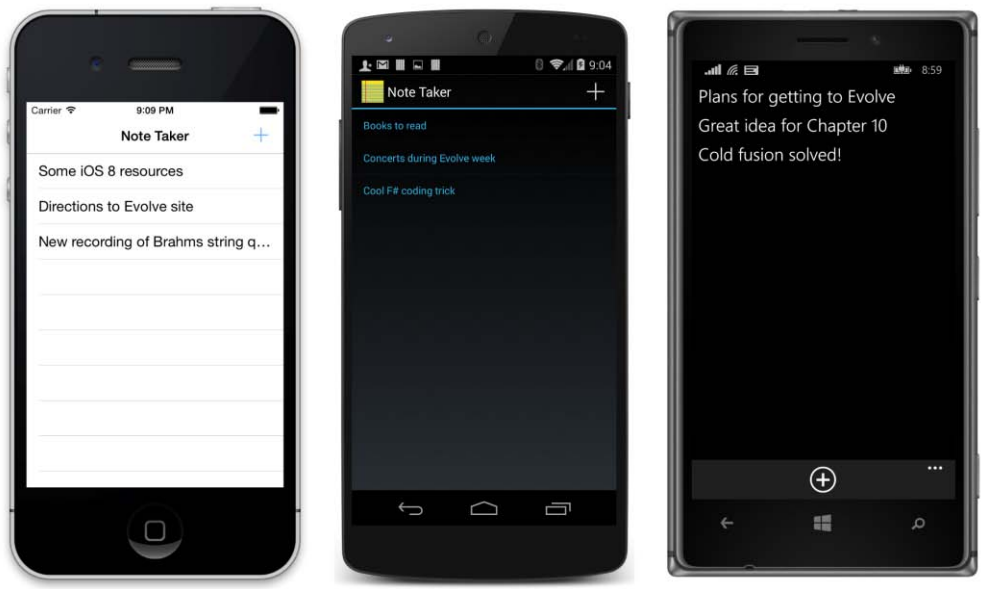
Building an app: Infrastructure

Sometimes programming tutorials such as this one can cover a lot of ground before directly addressing the needs of real-world applications. This chapter and the next are an attempt to avoid that common problem.

These two chapters present an actual application built from the ground up. This won't be a very sophisticated application, but it will be usable and useful. The application's name is **NoteTaker**, and it lets you take notes and save them.

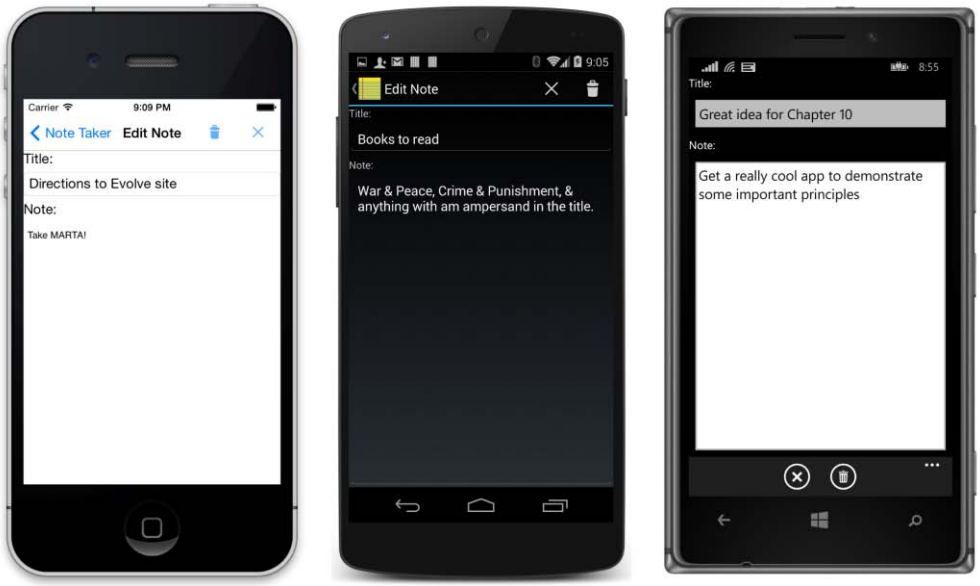
Each note is some text—as much text as you want—and an optional title. If you don't specify a title, one will be generated for you from the first few words of the note.

When the program is finally finished at the end of the next chapter, it will have the following home screen:



That's a scrollable lists of the titles of the notes you've already created. A **New** button symbolized by a plus sign appears as a toolbar item either at the top or bottom of the screen. Tap that button and the program navigates to a page for entering a new note:

You can also edit an existing note by tapping that note on the home screen:



Toolbar items allow you to cancel the editing or delete the note. The **New Note** and **Edit Note** screens are very similar, so it's probably not surprising that they are simply variations of the same page.

Engineering in general is a problem-solving activity, and programming is no different. We begin with a vision and then plan a strategy and set some goals to begin realizing that vision in code. But in attempting to build the program, problems and obstacles are often encountered. It is in solving these

problems that we battle through our ignorance to acquire knowledge and skills.

In this chapter the **NoteTaker** program goes through six different versions. They are numbered 1 through 6, but the version numbers should probably be more like 0.1 through 0.6. In the course of enhancing this program through these six versions, the following topics are encountered:

- The `Entry` and `Editor` views for editing text
- File input/output (I/O) in the mobile environment
- Asynchronous operations
- Property change notifications
- Data binding

The next chapter gets into multi-page architectures, page navigation, the powerful `ListView` for displaying collections of items, and dealing with application lifecycle issues.

Version 1. The Entry and Editor views

The **NoteTaker** app definitely requires some text input. Some phones have physical keyboards, but the vast majority are limited to virtual onscreen keyboards. Of course, these are somewhat different for each platform and often also vary by the type of text input.

Xamarin.Forms defines two views for obtaining text input:

- `Entry` for a single line of text
- `Editor` for multiple lines of text

Both `Entry` and `Editor` derive from `InputView`, which derives from `View`.

All three platforms have various styles of virtual keyboard appropriate for different types of text input. For example, a keyboard for typing a URL should be different from a keyboard for entering a phone number. For this reason, `InputView` defines a property named `Keyboard` of type `Keyboard`, a class that defines seven static read-only properties of type `Keyboard` appropriate for different keyboard uses:

- `Default`
- `Text`
- `Chat`
- `Url`
- `Email`

- `Numeric`
- `Telephone`

For example, if you have an `Entry` view intended for entering a URL, specify:

```
entry.Keyboard = Keyboard.Url;
```

Of course, the actual keyboards are platform-specific, and not all three platforms have distinct keyboards for all seven static properties.

The `Keyboard` class an alternative way to specify a keyboard using a `KeyboardFlags` enumeration, which has the following flag members:

- `CapitalizeSentence` (equal to 1)
- `Spellcheck` (2)
- `Suggestions` (4)
- `All` (\xFFFFFFFF)

These flags make more sense with an `Editor` rather than an `Entry` because the user is likely to be typing real text consisting of actual words organized into sentences. Set the keyboard like this:

```
editor.Keyboard = Keyboard.Create(KeyboardFlags.All);
```

The phone's operating system displays the virtual keyboard when the `Entry` or `Editor` acquires *keyboard input focus*. Input focus means that keyboard input is directed towards that particular view. Only one view can have input focus at any time. Although any Xamarin.Forms view can get input focus, only `Entry` and `Editor` are equipped to process that input.

A view must have its `IsEnabled` property set to `true` (the default state) to acquire input focus. A user can give an enabled view input focus by tapping it. When the `Entry` or `Editor` acquires input focus, the operating system pops up the keyboard.

A user can remove keyboard focus from an `Entry` or `Editor` by tapping somewhere else on the screen. The keyboard is automatically dismissed. But this is not the only way to dismiss the keyboard. Often a specific keyboard key or button associated with the keyboard lets the user signal that text input is complete. The concept of signaling that text is complete is complicated somewhat by the different nature of the `Entry` and `Editor` views: An **Enter** or **Return** key often removes input focus from an `Entry` view, but in an `Editor` that same key instead simply marks the end of one paragraph and the beginning of another.

On the iPhone, a **Return** button dismisses the keyboard from an `Entry` and a special **Done** button dismisses the keyboard from an `Editor`. On the Android, a **Done** key dismisses the keyboard from the `Entry` but a button on the bottom of the screen is required to dismiss the keyboard from the `Editor`. On the Windows Phone, the **Enter** key dismisses the keyboard from the `Entry` but the hardware **Back** button dismisses the keyboard from the `Editor`.

The `Entry` and `Editor` define `Focused` and `Unfocused` events that signal gaining and losing input focus. The `IsFocused` property indicates if a particular view currently has input focus. A program can attempt to set focus to a particular view in code by calling the `Focus` method on the view. The method returns `true` if the focus change was successful. These focus-related members are defined by `Visual-Element`, the base class for all UI elements, but they are really only relevant for the `Entry` and `Editor`.

Both `Entry` and `Editor` define `Text` properties that expose the current text displayed by the view. A program can initialize this `Text` property and then allow the user to edit that text.

Both `Entry` and `Editor` define a `Completed` event that is fired when the user has signaled that editing is completed. This is an excellent opportunity for programs to access the `Text` property and save the results.

Both `Entry` and `Editor` also define a `TextChanged` event that is fired when the `Text` property changes. (Keep in mind that .NET `string` objects are immutable. A `string` object can't itself change after it's been created. Every time the contents of the `Entry` or `Editor` change, it's a whole new `string` object rather than a modified `string` object.) The `TextChanged` event can be a valuable means for the application to monitor the text input on a character-by-character basis and respond to changes before the user signals that the typing is complete.

You might assume that `Entry` is somewhat simpler than `Editor` because it handles only a single line of text. However, `Entry` has three additional properties that `Editor` doesn't have:

- `TextColor` — a `Color` value
- `IsPassword` — a Boolean that causes characters to be masked
- `Placeholder` — light colored text that appears in the `Entry` but disappears as soon as the user begins typing.

The **NoteTaker** program requires a page that contains an `Entry` for the user to type a title for the note, and an `Editor` for typing the note itself. For the first version of this program—called **NoteTaker1**—let's not do much more beyond getting these two views on the page with a couple of `Label` views for identification.

The `Editor` allows an indefinite amount of text to be entered and internally implements scrolling. As you discovered with the `ScrollView` in the previous chapter, sharing a scrollable view with other views on the page requires that the `VerticalOptions` property be set to `LayoutOptions.FillAndExpand`.

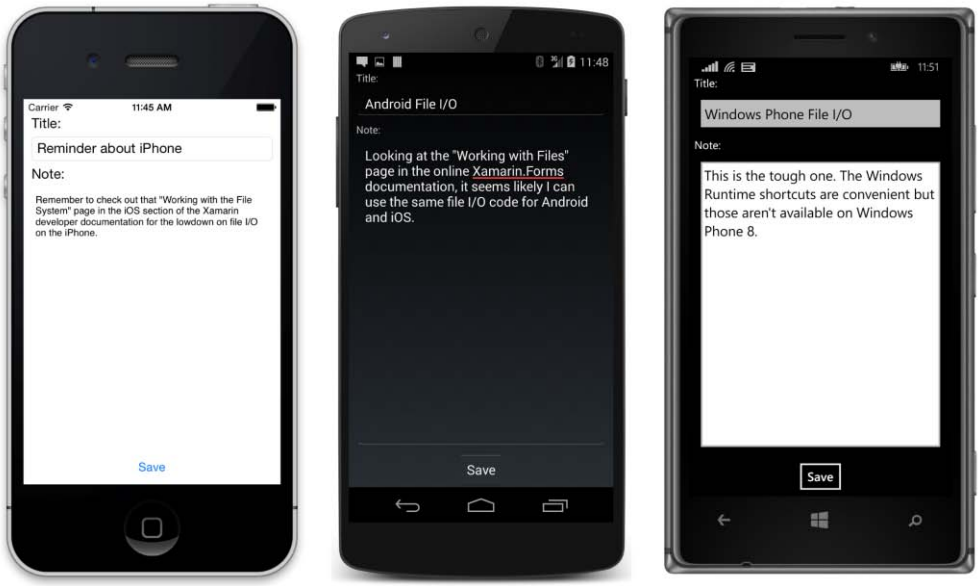
Here's the `NoteTaker1Page` class:

```
class NoteTaker1Page : ContentPage
{
    public NoteTaker1Page()
    {
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);
```

```
        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                new Entry
                {
                    Placeholder = "Title (optional)"
                },
                new Label
                {
                    Text = "Note:"
                },
                new Editor
                {
                    Keyboard = Keyboard.Create(KeyboardFlags.All),
                    BackgroundColor = Device.OnPlatform(Color.Default,
                                                        Color.Default,
                                                        Color.White),
                    VerticalOptions = LayoutOptions.FillAndExpand
                },
                new Button
                {
                    Text = "Save",
                    HorizontalOptions = LayoutOptions.Center
                }
            }
        };
    }
}
```

The program explicitly sets the `BackgroundColor` property on the `Editor` to correct a little flaw in the Windows Phone implementation: Without this setting, the `Editor` background goes black when it loses input focus, and that causes the black text to be invisible.

Here's what it looks like with some text typed in:



The `Button` at the bottom labeled **Save** currently does nothing. Such a button is undoubtedly intended to save the note to a file. To do that, however, we need to know how to save files, and later load them back in.

File input/output is not something included in the Xamarin.Forms API. Yet, some file I/O is required by many mobile apps, if only to save settings and interim data. For this reason, file I/O is probably the most compelling reason to skirt around the Xamarin.Forms API and implement some vital features of your app by using the programming interfaces of the individual platforms.

Version 2. File input/output

Traditionally, file input/output is one of the most basic programming tasks, but file I/O on mobile devices is a little different than on the desktop. On the desktop, users and applications generally have a whole disk available organized in a directory structure. On mobile devices, several standard folders exist—for pictures or music, for example—but application-specific data is generally restricted to private storage areas.

Programmers familiar with .NET know that the `System.IO` namespace contains the bulk of standard file I/O support. Perhaps the most important class in this namespace is the static `File` class, which not only provides a bunch of methods to create new files and open existing files but also includes several methods capable of performing an entire file read or write operation in a single method call.

For example, the `File.WriteAllText` method has two arguments of type `string`—a filename and the file contents. The method creates the file (replacing an existing file with the same name if

necessary), writes the contents to the file, and closes it. The `File.ReadAllText` method is similar but returns the contents of the file in one big `string` object. These methods seem ideal for the job of saving and retrieving notes.

The Xamarin.iOS and Xamarin.Android libraries include a version of .NET that has been expressly tailored by Xamarin for these two mobile platforms. The methods in the `File` class in the `System.IO` namespace map to appropriate file I/O functions in the iOS and Android platforms. This means that you can use methods in the `File` class—including `File.WriteAllText` and `File.ReadAll-Text`—in your iPhone and Android applications.

Let's experiment a bit:

Go into Visual Studio or Xamarin Studio, and load any Xamarin.Forms solution created so far, such as **NoteTaker1**. Bring up one of the code files in the iOS or Android project. In a constructor or method, type the `System.IO` namespace name and then a period. You'll get a list of all the available types in the namespace. If you then type `File` and a period, you'll get all the static methods in the `File` class, including `WriteAllText` and `ReadAllText`.

In a Windows Phone project, however, you're working with a version of .NET created by Microsoft and stripped down somewhat for the Windows Phone platform. If you type `System.IO.File` and a period, you'll see a rather diminished `File` class that does *not* include `WriteAllText` and `ReadAll-Text`, although it does include methods to create and open text files.

Now go into any code file in a Xamarin.Forms Portable Class Library project, and type `System.IO` and a period. Now you won't even see the `File` class! It does not exist in the PCL. Why is that? PCLs are configured to support multiple target platforms. The APIs implemented within the PCL are necessarily an intersection of the APIs in these target platforms.

A PCL appropriate for Xamarin.Forms includes the following platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone Silverlight 8
- Xamarin.Android
- Xamarin.iOS

Notice the inclusion of Windows 8, which incorporates an API called the Windows Runtime (or WinRT). Microsoft completely revamped file I/O for WinRT and created a whole new file I/O API. The `System.IO.File` class does not exist in the PCL because it is not part of WinRT.

Although the `File` class does not exist in a Portable Class Library project, you might wonder what kind of `File` class you can use in a Shared Asset Project. Well, it varies by what platform is being compiled. You can use `File.WriteAllText` and `File.ReadAllText` in your iOS and Android projects but not in your Windows Phone projects. Your Windows Phone projects need something else.

Skip past the scary stuff?

Already you might suspect that this subject of file I/O is going to get hairy, and you are correct. But consider what we're trying to do here: We're trying to target three different mobile platforms with a common code base. That's not easy, and we're bound to encounter some rough terrain along the way.

So the question has to be: Metaphorically speaking, do you enjoy hiking through treacherous terrain to climb to the top of a mountain and get a gorgeous view? Or would you prefer that somebody else takes a photo from the top of the mountain and sends it to you in an email?

If you're in the latter category, you might prefer to skip over much of this chapter and jump straight to the comparatively simple and elegant solution to the problem of Xamarin.Forms file I/O presented at the chapter's end and continuing into the next chapter. Until that point, some of the transitional code you'll encounter will be both scary and ugly.

If you choose to take this long path up the mountain, you'll understand why the trip is necessary, and you'll understand the rationale behind the platform differences—even in seemingly routine jobs like file I/O.

In the previous chapter you saw how the Xamarin.Forms `Device` class can be a valuable tool for dealing with platform differences. But the code that's referenced by the `Device` class must be compilable in all three platforms. This is not the case for file I/O because the different platforms have access to different APIs. This means that the platform differences can't be managed using the `Device` class and must be handled in other ways. Moreover, the solutions are different for Shared Asset Projects and Portable Class Libraries projects.

For this reason, for the next two versions of **NoteTaker**, there will be separate solutions for SAP and PCL. The two different solutions for version 2 are named **NoteTaker2Sap** and **NoteTaker2Pcl**.

Preprocessing in the SAP

Dealing with platform differences is a Shared Asset Project is a little more straightforward than a PCL and involves more traditional programming tools, so let's begin with that.

In code files in a Shared Asset Project, you can use the C# preprocessor directives `#if`, `#elif`, `#else`, and `#endif` with conditional compilation symbols defined for the three platforms. These symbols are __IOS__ for iOS and WINDOWS_PHONE for Windows Phone; there are no conditional compilation symbols for Android, but Android can be identified as not being iOS or Windows Phone.

The **NoteTaker2Sap** project includes a class named `FileHelper` in a file named FileHelper.cs. You can add such a file to the project the same way you add a new file for the class that derives from `ContentPage`.

The FileHelper.cs file uses C# preprocessor directives to divide the code into two sections. The first section is for iOS and Android and is compiled if the WINDOWS_PHONE identifier is not defined. The second section is for Windows Phone.

Both sections contain three public static methods named `Exists`, `WriteAllText`, and `ReadAllText`. In the first section, the iOS and Android versions of these functions use standard static `File` methods but with a folder name obtained from the `Environment.GetFolderPath` method with an argument of `Environment.SpecialFolder.MyDocuments`:

```
namespace NoteTaker2Sap
{
    static class FileHelper
    {

#if !WINDOWS_PHONE // iOS and Android

        public static bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public static void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public static string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        static string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }

#else // Windows Phone

        public static bool Exists(string filename)
        {
            return File.Exists(filename);
        }

        public static void WriteAllText(string filename, string text)
        {
            StreamWriter writer = File.CreateText(filename);
            writer.Write(text);
            writer.Close();
        }

        public static string ReadAllText (string filename)
        {
            StreamReader reader = File.OpenText(filename);
            string text = reader.ReadToEnd();
```

```
            reader.Close();
            return text;
        }

#endif

    }
}
```

When the Shared Asset Project is compiled for Windows Phone, the `File.WriteAllText` and `File.ReadAllText` methods do not exist so those can't appear in the Windows Phone section. However, static `CreateText` and `OpenText` methods are available, and these are used to obtain `StreamWriter` and `StreamReader` objects. This Windows Phone code works in the sense that it doesn't raise an exception, but you'll see shortly that it really doesn't do what you want. Something else is required.

Besides the `FileHelper` class to handle low-level file I/O, the **NoteTaker2Sap** project includes another new class named `Note`. This class encapsulates the two `string` objects associated with a note in simple read/write properties named `Title` and `Text`. This class also includes methods named `Save` and `Load` that call the appropriate methods in `FileHelper`:

```
namespace NoteTaker2Sap
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text);
        }

        public void Load(string filename)
        {
            string text = FileHelper.ReadAllText(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }
    }
}
```

Notice that the `Save` method simply joins the two `string` objects into one with a line feed character, and the `Load` method takes them apart.

Finally, the `NoteTaker2SapPage` class has a very similar page layout as the first program but contains two buttons labeled **Save** and **Load** that use the `Note` class for these operations. A filename

of “test.note” is used throughout:

```
class NoteTaker2SapPage : ContentPage
{
    static readonly string FILENAME = "test.note";

    Entry entry;
    Editor editor;
    Button loadButton;

    public NoteTaker2SapPage()
    {
        // Create Entry and Editor views.
        entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Create Save and Load buttons.
        Button saveButton = new Button
        {
            Text = "Save",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        saveButton.Clicked += OnSaveButtonClicked;

        loadButton = new Button
        {
            Text = "Load",
            IsEnabled = FileHelper.Exists(FILENAME),
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        loadButton.Clicked += OnLoadButtonClicked;

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
```

```
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }

    void OnSaveButtonClicked(object sender, EventArgs args)
    {
        Note note = new Note
        {
            Title = entry.Text,
            Text = editor.Text
        };
        note.Save(FILENAME);
        loadButton.IsEnabled = true;
    }

    void OnLoadButtonClicked(object sender, EventArgs args)
    {
        Note note = new Note();
        note.Load(FILENAME);
        entry.Text = note.Title;
        editor.Text = note.Text;
    }
}
```

Notice that the **Load** button initialization calls `FileHelper.Exists` to determine if the file exists and disables the button if it does not. The button is then enabled the first time the file is saved.

You'll want to convince yourself that this works—that the information is saved to the device (or phone simulator). Type something into the `Entry` and `Editor`, and then press **Save** to save that information. Clear out the `Entry` and `Editor` (or type in new text), and press **Load** to restore the information that was saved.

Here's what's really important: If you terminate the program or shut down the phone and then rerun the program, the saved file still exists.

Well, in two out of three cases, the saved file still exists. It works with iOS and Android, but not Windows Phone. Although the Windows Phone **Save** and **Load** buttons seem to work while the

program is running, the file is not persisted when the application is exited. Getting Windows Phone to work right will require a different set of file I/O classes.

Meanwhile, let's try to get this simple (two-thirds functional) version to work with a Portable Class Library solution.

Dependency service in the PCL

As you've seen, the `System.IO.File` class does not exist in the version of .NET available to a Xamarin.Forms Portable Class Library. This means that if you've created a PCL-based Xamarin.Forms solution, the file I/O code cannot be in the PCL. The file I/O code must be implemented in the individual platform projects where it can use the version of .NET specifically for that platform.

Yet, the PCL must somehow make calls to these file I/O functions. Normally that wouldn't work: Application projects make calls to libraries all the time, but libraries generally can't make calls to applications except with events or callback functions.

It is the main purpose of the Xamarin.Forms `DependencyService` class to get around this restriction. Although this class is implemented in the Xamarin.Forms.Core library assembly and used in a PCL, it uses .NET reflection to search through all the other assemblies available in the application, including the particular platform-specific application assembly itself. (It is also possible for the platform projects to use dependency injection techniques to configure the PCL to make calls into the platform projects.)

To use `DependencyService`, the first requirement is that the PCL must contain an interface definition that includes the names and signatures of the platform-specific methods you need. Here is that file in the **NoteTaker2Pcl** project:

```
namespace NoteTaker2Pcl
{
    public interface IFileHelper
    {
        bool Exists(string filename);

        void WriteAllText(string filename, string text);

        string ReadAllText(string filename);
    }
}
```

This interface must be public to the PCL because it must be visible to the individual platform projects.

Next, in all three application projects, you create code files with classes that implement this interface. Here's the one in the iOS project. (You can tell that this file is in the iOS project by the namespace):

```
using System;
using System.IO;
using Xamarin.Forms;
```

```
[assembly: Dependency(typeof(NoteTaker2Pcl.iOS.FileHelper))]

namespace NoteTaker2Pcl.iOS
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(
                        Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

The actual implementation of these methods involves the same code that you've already seen except with instance methods rather than static methods. But take note of two necessary characteristics of this file:

- The class implements the `IFileHelper` interface defined in the PCL. Because it implements that interface, the three methods defined in the interface must be defined as public in this class.
- A special assembly-level attribute named `Dependency` is defined prior to the `namespace` definition.

`Dependency` is a special Xamarin.Forms attribute defined by the `DependencyAttribute` class specifically for use with the `DependencyService` class. The `Dependency` attribute simply specifies the type of the class but it assists the `DependencyService` class in locating the implementation of the interface in the application projects.

A similar file is in the Android project:

```
using System;
using System.IO;
```

```
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker2Pcl.Droid.FileHelper))]

namespace NoteTaker2Pcl.Droid
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(
                        Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

And in the Windows Phone project:

```
using System;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker2Pcl.WinPhone.FileHelper))]

namespace NoteTaker2Pcl.WinPhone
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            return File.Exists(filename);
        }

        public void WriteAllText(string filename, string text)
        {
            StreamWriter writer = File.CreateText(filename);
```

```
            writer.Write(text);
            writer.Close();
        }

        public string ReadAllText(string filename)
        {
            StreamReader reader = File.OpenText(filename);
            string text = reader.ReadToEnd();
            reader.Close();
            return text;
        }
    }
}
```

Now the hard work is done. You'll recall that the `Note` class in **NoteTaker2Sap** made calls to `FileHelper.WriteAllText` and `FileHelper.ReadAllText`. The `Note` class in **NoteTaker2Pcl** is very similar but instead references the two methods through the static `DependencyService.Get` method. This is a generic method that requires the interface as a generic argument but then is capable of calling any method in that interface:

```
namespace NoteTaker2Pcl
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            DependencyService.Get<IFileHelper>().WriteAllText(filename, text);
        }

        public void Load(string filename)
        {
            string text = DependencyService.Get<IFileHelper>().ReadAllText(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }
    }
}
```

Internally, the `DependencyService` class searches for the interface implementation in the particular platform project and makes a call to the specified method.

The `NoteTaker2PclPage` class is nearly the same as `NoteTaker2SapPage` except it also uses `DependencyService.Get` to call the `Exists` method during the initialization of the **Load** button.

```
    loadButton = new Button
```

```
{
    Text = "Load",
    IsEnabled = DependencyService.Get<IFileHelper>().Exists(FILENAME),
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
```

Of course, the **NoteTaker2Pcl** version has the same deficiency as **NoteTaker2Sap** in that it doesn't persist the data for Windows Phone.

Version 3. Going async

The data isn't persisted for Windows Phone because the file I/O code is not correct. A Xamarin.Forms application targets Windows Phone 8, which implements a subset of the same WinRT file I/O available to Windows 8 applications, largely found in the new `Windows.Storage` and `Windows.Storage.Streams` namespaces.

Windows Phone 8 continues to support some older file I/O functions that Windows Phone 7 inherited from Silverlight, but these are not recommended for new Windows Phone 8 applications. Windows Phone 8 applications should instead use the WinRT file I/O API, and the programs in this book follow that recommendation.

Part of the impetus behind this new array of file I/O classes in Windows 8 and Windows Phone 8 is a recognition of a transition away from the relatively unconstrained file access of desktop applications towards a more sandboxed environment. To store data that is private to an application, a Windows Phone program first gets a special `StorageFolder` object:

```
StorageFolder localFolder = ApplicationData.Current.LocalFolder;
```

`ApplicationData` has a static property named `Current` that returns the `ApplicationData` object for the application. `LocalFolder` is an instance property of `ApplicationData`.

`StorageFolder` defines methods named `CreateFileAsync` to create a new file and `GetFileAsync` to open an existing file. These two methods return objects of type `StorageFile`. At this point, a program can open the file for writing or reading with `OpenAsync` or `OpenReadAsync`. These methods return an `IRandomAccessStream` object. From this, `DataWriter` or `DataReader` objects are created to perform write or read operations.

This sounds a bit lengthy, and it is. A rather simpler approach for text files involves the static methods `FileIO.ReadTextAsync` and `FileIO.WriteTextAsync`. The first argument to these methods is a `StorageFile` object, and the methods incorporate all the operations to open the file, write to it or read from it, and close the file. Although these methods are available in Windows Phone 8.1, they are not in Windows Phone 8, the version of Windows Phone supported by Xamarin.Forms.

At any rate, by this time you've undoubtedly noticed the frequent `Async` suffix on these method names. These are asynchronous methods. Internally, these methods spin off secondary threads of execution for doing the actual work and return quickly to the caller. The work takes place in the

background, and when that work is finished—when the file has been created or opened, or written to or read from—the caller is notified through a call-back function and provided with the function's result.

Which might raise the question: *Why* are these methods asynchronous? Why are they more complex than the old .NET file I/O functions?

Graphical user interfaces have an intrinsic problem. Although an application can consist of multiple threads of execution, access to the user interface must usually be restricted to a single thread. The problem is not so much the graphical output, but the user input, which in various environments might include the keyboard, mouse, pen, or touch. In general, it can't be known where a user input event should be routed until all previous user input events have been processed. This means that user input events must be processed sequentially in a single thread of execution.

The impact of this restriction is profound: In the general case, *all* the application's user interface processing must be handled in a single thread, often called the *UI thread*. Even the seemingly innocent act of using a secondary thread of execution to set a property of a user-interface object such as `Entry` or `Editor` is forbidden.

At the same time, programmers are cautioned against doing any lengthy processing in this UI thread. If the UI thread is carrying out some lengthy processing, it can't respond to user input events and the entire user interface can seem to freeze up.

As we users have become more accustomed to graphical user interfaces over the decades, we've become increasingly intolerant of even the slightest lapse in responsiveness. Consequently, as application programmers, we are increasingly encouraged to avoid lengthy processing in the UI thread and to keep the application as responsive as possible.

This implies that lengthy processing jobs should be relegated to secondary threads of execution. These threads run "in the background," asynchronously with the UI thread.

The future of computing will undoubtedly involve a lot more asynchronous computing and parallel processing, particularly with the increasing use of multicore processor chips. Developers will need good language tools to work with asynchronous operations, and fortunately C# has been in the forefront in this regard.

When the WinRT APIs used for Windows 8 Store apps were being developed, the Microsoft developers took a good hard look at timing and decided that any function call that could require more than 50 milliseconds to execute should be made asynchronous so that it would not interfere with the responsiveness of the user interface.

APIs that require more than 50 milliseconds obviously include file I/O functions, which often need to access potentially slow pieces of hardware, like disk drives or a network. Any WinRT file I/O function that could possibly hit a physical storage device was made asynchronous and given an `Async` method suffix.

The `CreateFileAsync` method defined by the `StorageFolder` class does not directly return a `StorageFile` object. Instead, it returns an `IAsyncOperation<StorageFile>` object:

```
IAsyncOperation<StorageFile> createOp = storageFolder.CreateFileAsync("filename");
```

The `IAsyncOperation` interface, its base interface `IAsyncInfo`, and related interfaces such as `IAsyncAction`, are all defined in the `Windows.Foundation` namespace, indicating how fundamental they are to the entire operating system. A return value such as `IAsyncOperation` is sometimes referred to as a "promise." The `StorageFile` object is not available just yet, but it will be in the future if nothing goes awry.

To begin the actual asynchronous operation, you must assign a handler to the `Completed` property of the `IAsyncOperation` object:

```
createOp.Completed = OnCreateFileCompleted;
```

`Completed` is a property rather than an event but it functions much like an event. The big difference is that `Completed` can't have multiple handlers. Assigning a callback method (named `OnCreateFile-Completed` in this example) actually initiates the background process.

The code that sets the `Completed` property to a handler executes very quickly, and the program can then continue normally. Simultaneously, the file is being created in a secondary thread. When the file is created, that background code calls the callback method assigned to the `Completed` handler in your code. That callback method might look like this:

```
void OnCreateFileCompleted(IAsyncOperation<StorageFile> createOp, AsyncStatus asyncStatus)
{
    if (asyncStatus == AsyncStatus.Completed)
    {
        StorageFile storageFile = createOp.GetResults();
        // continue with next step ...
    }
    else
    {
        // deal with cancellation or error
    }
}
```

The second argument indicates the status, and at this point it's either `Completed`, `Canceled`, or `Error`. Members of the first argument can provide more detail about any error that might have occurred. If all is well, calling `GetResults` on the first argument obtains the `StorageFile` object.

At this point, the next step would be to open that file for writing. A call to `OpenAsync` returns an object of type `IAsyncOperation<IRandomAccessStream>`, and that involves another callback method:

```
void OnCreateFileCompleted(IAsyncOperation<StorageFile> createOp, AsyncStatus asyncStatus)
{
    if (asyncStatus == AsyncStatus.Completed)
    {
        StorageFile storageFile = createOp.GetResults();
```

```
        IAsyncOperation<IRandomAccessStream> openOp =
                    storageFile.OpenAsync(FileAccessMode.ReadWrite);
        openOp.Completed = OnOpenFileCompleted;
    }
    else
    {
        // deal with cancellation or error
    }
}

void OnOpenFileCompleted(IAsyncOperation<IRandomAccessStream> openOp, AsyncStatus asyncStatus)
{
    // ...
}
```

One way to simplify this code is to use anonymous lambda functions for the callbacks. This avoids a proliferation of individual methods and allows more free-form access to local variables. But for a sequence of asynchronous method calls, it tends to produce a nested structure of asynchronous callbacks, as you'll see.

Asynchronous lambdas in the SAP

In the **NoteTaker3Sap** project, the file I/O code has been moved to the `Note` class and performed separately for Windows Phone using lambda functions for callbacks. To keep the code simple (at least comparatively so), there is no error handling:

```
using System;

#if !WINDOWS_PHONE
using System.IO;
#else
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
#endif

namespace NoteTaker3Sap
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;

#if !WINDOWS_PHONE // iOS and Android

            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

```
            string filepath = Path.Combine(docsPath, filename);
            File.WriteAllText(filepath, text);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp =
                            localFolder.CreateFileAsync(filename,
                                CreationCollisionOption.ReplaceExisting);

            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStream> openOp =
                                        storageFile.OpenAsync(FileAccessMode.ReadWrite);
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataWriter dataWriter = new DataWriter(stream);
                    dataWriter.WriteString(text);
                    DataWriterStoreOperation storeOp = dataWriter.StoreAsync();
                    storeOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
                        dataWriter.Dispose();
                    };
                };
            };

#endif
        }

        public void Load(string filename)
        {

#if !WINDOWS_PHONE // iOS and Android

            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            string filepath = Path.Combine(docsPath, filename);
            string text = File.ReadAllText(filepath);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                        storageFile.OpenReadAsync();
```

```
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataReader dataReader = new DataReader(stream);
                    uint length = (uint)stream.Size;
                    DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
                    loadOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
                        string text = dataReader.ReadString(length);
                        dataReader.Dispose();

                        // Break string into Title and Text.
                        int index = text.IndexOf('\n');
                        this.Title = text.Substring(0, index);
                        this.Text = text.Substring(index + 1);
                    };
                };
            };

#endif

        }
    }
}
```

Notice the `Dispose` calls on the `DataWriter` and `DataReader` methods. It might be tempting to remove these calls under the assumption that the objects are disposed automatically when the objects go out of scope, but this is not the case. If `Dispose` is not called, the files remain open,

But the big question is: Why has all this code been moved to the `Note` class? Why isn't it isolated in a separate `FileHelper` class as in the previous version of the program?

The problem is that a method that requires asynchronous callbacks to obtain an object can't directly return that object to the caller. Look at the `Load` method here. When that `Load` method is called in a Windows Phone program, the `localFolder` variable is set, and the `createOp` object is set, but as soon as the `Completed` property is set to the asynchronous callback method, the `Load` method returns to the caller. But the method doesn't yet have anything to return! The `GetFileAsync` operation is proceeding in the background in a secondary thread. Only later does the `Completed` callback method execute for the next step of the job. When the contents of the file are finally read within these nested callbacks, the contents must be stored somewhere. Fortunately, the code is in the `Note` class, so the results can be stored in the `Title` and `Text` properties.

In the previous version of this program, the `Exists` method returned a Boolean to indicate the existence of a file. That code needs to be moved to the `NoteTaker3SapPage` class where it has access to the `Button` whose `IsEnabled` property must be set:

```
using System;

#if !WINDOWS_PHONE
using System.IO;
#else
```

```
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
#endif

using Xamarin.Forms;

namespace NoteTaker3Sap
{
    class NoteTaker3SapPage : ContentPage
    {
        static readonly string FILENAME = "test.note";

        Note note = new Note();
        Entry entry;
        Editor editor;
        Button loadButton;

        public NoteTaker3SapPage()
        {
            // Create Entry and Editor views.
            entry = new Entry
            {
                Placeholder = "Title (optional)"
            };

            editor = new Editor
            {
                Keyboard = Keyboard.Create(KeyboardFlags.All),
                BackgroundColor = Device.OnPlatform(Color.Default,
                                                    Color.Default,
                                                    Color.White),
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Create Save and Load buttons.
            Button saveButton = new Button
            {
                Text = "Save",
                HorizontalOptions = LayoutOptions.CenterAndExpand
            };
            saveButton.Clicked += OnSaveButtonClicked;

            loadButton = new Button
            {
                Text = "Load",
                IsEnabled = false,
                HorizontalOptions = LayoutOptions.CenterAndExpand
            };
            loadButton.Clicked += OnLoadButtonClicked;

            // Check if the file is available.

#if !WINDOWS_PHONE // iOS and Android
```

```
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            string filepath = Path.Combine(docsPath, FILENAME);
            loadButton.IsEnabled = File.Exists(filepath);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(FILENAME);
            createOp.Completed = (asyncInfo, asyncStatus) =>
            {
                loadButton.IsEnabled = asyncStatus != AsyncStatus.Error;
            };

#endif

            // Assemble page.
            this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

            this.Content = new StackLayout
            {
                Children =
                {
                    new Label
                    {
                        Text = "Title:"
                    },
                    entry,
                    new Label
                    {
                        Text = "Note:"
                    },
                    editor,
                    new StackLayout
                    {
                        Orientation = StackOrientation.Horizontal,
                        Children =
                        {
                            saveButton,
                            loadButton
                        }
                    }
                }
            };
        }

        void OnSaveButtonClicked(object sender, EventArgs args)
        {
            note.Title = entry.Text;
            note.Text = editor.Text;
            note.Save(FILENAME);
            this.loadButton.IsEnabled = true;
        }
```

```
        void OnLoadButtonClicked(object sender, EventArgs args)
        {
            note.Load(FILENAME);
            entry.Text = note.Title;
            editor.Text = note.Text;
        }
    }
}
```

To set that `IsEnabled` property of the **Load** button in the Windows Phone version, the strategy is to attempt to call `GetFileAsync`. If that call reports an error in the asynchronous callback, the file does not exist. (The `StorageFile` class defines an `IsAvailable` property, but it isn't supported on Windows Phone.)

Notice that this version of the page class contains a single `Note` object instantiated as a field and accessed by both `Button` event handlers. This makes more sense than creating a new `Note` object in each call to the `Clicked` handlers. In the final version of the program, when a page like this is used for creating a new note or editing an existing note, the page and the `Note` object will exist as a tightly-linked pair—one class for the user interface, and another class for the underlying data exposed by the user interface.

If you try out this program on Windows Phone, you'll still discover a problem:

Type some text in the `Entry` and `Editor`. Press the **Save** button. Now erase that text or change it. Press the **Load** button. The saved text returns. Great!

Now end the program and start it up again. The `Entry` and `Editor` fields are blank but the **Load** button is enabled. Excellent! The file still exists. Press the **Load** button. Nothing. That's odd: If the `Button` is enabled, the file should exist, so where is it? Now press the **Load** button a second time. Ahh, there it is!

Can you figure out what's happening?

When you run the program and press the **Load** button to load a previously created file, the **Load** method in `Note` is called. But that method returns after calling `GetFileAsync`. The file hasn't been read yet, the `Title` and `Text` properties of `Note` haven't yet been set, but the `OnLoadButton-Clicked` method blithely sets the contents of those `Title` and `Text` properties to the `Entry` and `Editor`. The callbacks in the `Load` method in `Note` continue to execute until the file is read and the `Title` and `Text` properties are eventually set, so pressing the **Load** button a second retrieves them.

This problem shows up only when the program starts up, because thereafter the values in the `Note` object are always the last values saved to the file.

Method callbacks in the PCL

Can these asynchronous method calls be incorporated in a PCL project that uses the `Dependency-Service` to access platform-specific versions of the file I/O logic? Certainly not in the same form as in the SAP version. The `Exists` and `ReadAllText` methods must return values—a `bool` and a `string`,

respectively—and we've already seen that the asynchronous function calls in a method can't return values.

But these methods *can* return values if they return those values in their own callback functions!

Here's the new `IFileHelper` interface in the **NoteTaker3Pcl** project:

```
using System;

namespace NoteTaker3Pcl
{
    public interface IFileHelper
    {
        void Exists(string filename, Action<bool> completed);

        void WriteAllText(string filename, string text, Action completed);

        void ReadAllText(string filename, Action<string> completed);
    }
}
```

All three methods now have a return type of `void`, but they all have a last argument that is a delegate for a method with (respectively) one Boolean argument, no arguments, and one `string` argument.

The iOS implementation of this interface is very similar to the previous version except that the `completed` method is called to indicate completion and to return any value:

```
using System;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker3Pcl.iOS.FileHelper))]

namespace NoteTaker3Pcl.iOS
{
    class FileHelper : IFileHelper
    {
        public void Exists(string filename, Action<bool> completed)
        {
            bool exists = File.Exists(GetFilePath(filename));
            completed(exists);
        }

        public void WriteAllText(string filename, string text,
                                 Action completed)
        {
            File.WriteAllText(GetFilePath(filename), text);
            completed();
        }

        public void ReadAllText(string filename, Action<string> completed)
        {
            string text = File.ReadAllText(GetFilePath(filename));
            completed(text);
```

```
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

The Android version is very similar. The Windows Phone version has methods that call the `completed` function in the innermost nested asynchronous callback:

```
using System;
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker3Pcl.WinPhone.FileHelper))]

namespace NoteTaker3Pcl.WinPhone
{
    class FileHelper : IFileHelper
    {
        public void Exists(string filename, Action<bool> completed)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
            createOp.Completed = (asyncInfo, asyncStatus) =>
            {
                completed(asyncStatus != AsyncStatus.Error);
            };
        }

        public void WriteAllText(string filename, string text, Action completed)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp =
                        localFolder.CreateFileAsync(filename,
                            CreationCollisionOption.ReplaceExisting);
            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStream> openOp =
                        storageFile.OpenAsync(FileAccessMode.ReadWrite);
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataWriter dataWriter = new DataWriter(stream);
                    dataWriter.WriteString(text);
                    DataWriterStoreOperation storeOp = dataWriter.StoreAsync();
                    storeOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
```

```
                    dataWriter.Dispose();
                    completed();
                };
            };
        };
    }

    public void ReadAllText(string filename, Action<string> completed)
    {
        StorageFolder localFolder = ApplicationData.Current.LocalFolder;
        IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
        createOp.Completed = (asyncInfo1, asyncStatus1) =>
        {
            IStorageFile storageFile = asyncInfo1.GetResults();
            IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                    storageFile.OpenReadAsync();
            openOp.Completed = (asyncInfo2, asyncStatus2) =>
            {
                IRandomAccessStream stream = asyncInfo2.GetResults();
                DataReader dataReader = new DataReader(stream);
                uint length = (uint)stream.Size;
                DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
                loadOp.Completed = (asyncInfo3, asyncStatus3) =>
                {
                    string text = dataReader.ReadString(length);
                    dataReader.Dispose();
                    completed(text);
                };
            };
        };
    }
  }
}
```

To hide away the calls to the `DependencyService.Get` method, another file has been added to the **NoteTaker3Pcl** project. This class lets other code in the program use normal-looking static `FileHelper` methods:

```
namespace NoteTaker3Pcl
{
    static class FileHelper
    {
        static IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public static void Exists(string filename, Action<bool> completed)
        {
            fileHelper.Exists(filename, completed);
        }

        public static void WriteAllText(string filename, string text, Action completed)
        {
            fileHelper.WriteAllText(filename, text, completed);
        }
```

```
        public static void ReadAllText(string filename, Action<string> completed)
        {
            fileHelper.ReadAllText(filename, completed);
        }
    }
}
```

Notice that the class also saves the `DependencyService` object associated with the `IFileHelper` interface in a static field to make the actual calls more efficient. Although this class seems to implement the `IFileHelper` interface, it actually does not implement that interface because the class and methods are all static.

The `Note` class is now almost as simple as the original version. The only real difference is a `Load` method that sets the `Title` and `Text` fields in a lambda function passed to the `FileHelper.ReadAllText` method:

```
namespace NoteTaker3Pcl
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text, () => { });
        }

        public void Load(string filename)
        {
            FileHelper.ReadAllText(filename, (string text) =>
                {
                    // Break string into Title and Text.
                    int index = text.IndexOf('\n');
                    this.Title = text.Substring(0, index);
                    this.Text = text.Substring(index + 1);
                });
        }
    }
}
```

The only difference in the page file is the code to determine if the **Load** button should be disabled. The `IsEnabled` setting occurs in a lambda function passed to the `FileHelper.Exists` method:

```
loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
loadButton.Clicked += OnLoadButtonClicked;
```

```
// Check if the file is available.
FileHelper.Exists(FILENAME, (exists) =>
    {
        loadButton.IsEnabled = exists;
    });
```

Does this fix the problem with the first press of the **Load** button on the Windows Phone? No, it does not. The `OnLoadButtonClicked` method is still setting the `Entry` and `Editor` to text from `Note` class properties before that text has been loaded from the file. To work properly, that code would need to know when the `Note` properties were set before transferring them to the `Editor` and `Entry`. Or the page class would need to know when the `Title` and `Text` properties of the `Note` object changed values.

The basic problem involves the existence of properties that change value without notifying anybody of the change. Can we do something about this?

Version 4. I will notify you when the property changes

Let's step back a moment.

So far, all the versions of the program have contained a class deriving from `ContentPage` that displays a user interface allowing a user to enter and edit two pieces of text. These two pieces of text are also stored in a class named `Note`. This `Note` class stores data that underlies the user interface of the page class.

These two classes are really two sides of the same data—one class presents the data for editing by the user, and the other class handles the more low-level chores, including loading and saving the data in the file system.

Optimally, at any time, both classes should be dealing with the same data. But this is not the case. The page class doesn't know when the data in the `Note` class has changed, and the `Note` class doesn't know when the text in the `Entry` and `Editor` views has changed.

Keeping user interfaces in synchronization with underlying data is a common problem, and standard solutions are available to fix that problem. One of the most important is an interface named `INotifyPropertyChanged`. It's defined in the .NET `System.ComponentModel` namespace like so:

```
interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

The entire interface consists of just one event named `PropertyChanged`, but this event provides a simple universal way for a class to notify any other class that might be interested when one of its properties has changed values.

The `PropertyChangedEventHandler` delegate associated with the `PropertyChanged` event incorporates an event argument of `PropertyChangedEventArgs`. This class defines a public property of type `string` named `PropertyName` that identifies the property being changed.

What's that? A property named `PropertyName` that identifies the property being changed? Yes, it sounds a little confusing, but in practice it's quite simple.

The following **NoteTaker4** program was created with the PCL template, but a Shared Asset Project could implement these changes as well.

A class such as `Note` can implement the `INotifyPropertyChanged` interface by simply indicating that the class derives from that interface and including a public event of the correct type and name:

```
class Note : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    …

}
```

In theory, that's all that's required. However, a class that implements this interface should also actually fire the event whenever one of its public properties changes value. The `PropertyChanged-EventArgs` object accompanying the event identifies the actual property that's changed value. The property should have been assigned its new value by the time it fires the event.

In the previous versions of the `Note` class, the properties were defined with implicit backing fields:

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

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

Now they're going to need explicit private backing fields:

```
string title, text;
```

Here's the new definition of the `Title` property. The `Text` property is similar:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;

            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs("Title"));
            }
        }
```

```
    }
    get
    {
        return title;
    }
}
```

This is very standard `INotifyPropertyChanged` code. The `set` accessor begins by checking if the private field is the same as the incoming string, and only continues if it's not. Some programmers new to `INotifyPropertyChanged` want to skip this check, but it's important. The interface is called `INotifyPropertyChanged` and not `INotifyMaybePropertyChangedMaybeNot`. In some cases, failing to check if the property is actually changing can cause infinite recursion.

The `set` accessor continues by saving the new value in the backing field and only then firing the event.

Here's the complete `Note` class:

```
using System.ComponentModel;

namespace NoteTaker4
{
    class Note : INotifyPropertyChanged
    {
        string title, text;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Title
        {
            set
            {
                if (title != value)
                {
                    title = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("Title"));
                    }
                }
            }
            get
            {
                return title;
            }
        }

        public string Text
        {
            set
            {
                if (text != value)
```

```
                {
                    text = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("Text"));
                    }
                }
            }
            get
            {
                return text;
            }
        }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text, () => { });
        }

        public void Load(string filename)
        {
            FileHelper.ReadAllText(filename, (string text) =>
                {
                    // Break string into Title and Text.
                    int index = text.IndexOf('\n');
                    this.Title = text.Substring(0, index);
                    this.Text = text.Substring(index + 1);
                });
        }
    }
}
```

The various `FileHelper` classes are the same as those in **NoteTaker3Pcl**.

The `NoteTaker4Page` class defines an instance of `Note` as a field (as in the previous version of the program), but now the constructor also attaches a handler for the `PropertyChanged` event now defined by `Note`:

```
note.PropertyChanged += (sender, args) =>
    {
        switch (args.PropertyName)
        {
            case "Title":
                entry.Text = note.Title;
                break;

            case "Text":
                entry.Text = note.Text;
                break;
        }
    };
```

This could be a named event handler of course, and it could use an `if` and `else` rather that a `switch` and `case` to identify the property being changed. It then sets the new value of the property to the `Text` property of either the `Entry` or `Editor`.

It looks fine, but it still won't work on the Windows Phone. When you tap the **Load** button you'll get an Unauthorized Access exception. Now what's wrong?

Here's the problem: In the general case, callbacks from asynchronous methods do *not* execute in the same thread as the code that the initiated the operation. Instead, the callback executes in the background thread that carried out the asynchronous operation.

Let's follow it through: When you press the **Load** button, the Windows Phone `ReadAllText` method executes. When the text is obtained, it calls the `completed` method but in a secondary thread of execution. In the `Load` method in `Note`, that `completed` method sets the `Title` and `Text` properties. The new `Title` property causes a `PropertyChanged` event to fire, and in that handler the new `Title` property is set to the `Text` property of the `Entry` view.

Therefore, the `Entry` view is being accessed from a thread other than the user-interface thread, and that's not allowed. That's why the exception is raised.

Fortunately, the fix for this problem is fairly easy. The `Device` class has a `BeginInvokeOn-MainThread` method with an argument of type `Action`. Simply enclose the code you want to execute in the UI thread in the body of that `Action` argument. It's easiest to wrap the entire `switch` and `case` in that callback:

```
note.PropertyChanged += (sender, args) =>
    {
        Device.BeginInvokeOnMainThread(() =>
            {
                switch (args.PropertyName)
                {
                    case "Title":
                        entry.Text = note.Title;
                        break;

                    case "Text":
                        editor.Text = note.Text;
                        break;
                }
            });
    };
```

The `Device.BeginInvokeOnMainThread` effectively waits until the UI thread gets a time slice from the operating system's thread scheduler, and then it runs the specified code.

With this change, you'll find that when you rerun the Windows Phone app, you can press **Load** just once and the `Entry` and `Editor` will be set with the saved values. They're being set not in the handler for the **Load** button but in the `PropertyChanged` handler when the properties are actually updated with the values loaded from the file.

You can also go the other way and keep the `Note` class updated with the current values of the `Entry` and `Editor` views. Simply install `TextChanged` handlers:

```
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };

editor.TextChanged += (sender, args) =>
    {
        note.Text = args.NewTextValue;
    };
```

Wait a minute. Have we messed this up? The `PropertyChanged` handler is setting the `Entry` and `Editor` text from the `Note` properties, and now these two `TextChanged` handlers are setting the `Note` properties from the `Entry` and `Editor` text. Isn't that an infinite loop?

No, because `Entry`, `Editor`, and `Note` fire `Changed` events only when the property is actually changing. The potentially infinite loop is truncated when the corresponding properties are the same.

Now that the `Entry` and `Editor` views are kept consistent with the `Note` class, it's not necessary to set the `Note` object from the `Entry` and `Editor` in the `Save` handler. Nor do we need to set the `Entry` and `Editor` from the `Note` object in the `Load` handler. Here's the complete `NoteTaker4Page` class. Notice that the `Entry` and `Editor` instances no longer need to be saved as fields because they're no longer referenced in the `Clicked` handlers:

```
class NoteTaker4Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    Note note = new Note();
    Button loadButton;

    public NoteTaker4Page()
    {
        // Create Entry and Editor views.
        Entry entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        Editor editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Create Save and Load buttons.
        Button saveButton = new Button
```

```
{
    Text = "Save",
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
saveButton.Clicked += OnSaveButtonClicked;

loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
loadButton.Clicked += OnLoadButtonClicked;

// Check if the file is available.
FileHelper.Exists(FILENAME, (exists) =>
    {
        loadButton.IsEnabled = exists;
    });

// Handle the Note's PropertyChanged event.
note.PropertyChanged += (sender, args) =>
{
    Device.BeginInvokeOnMainThread(() =>
        {
            switch (args.PropertyName)
            {
                case "Title":
                    entry.Text = note.Title;
                    break;

                case "Text":
                    editor.Text = note.Text;
                    break;
            }
        });
};

// Handle the Entry and Editor TextChanged events.
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };

editor.TextChanged += (sender, args) =>
    {
        note.Text = args.NewTextValue;
    };

// Assemble page.
this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

this.Content = new StackLayout
{
```

```
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }

    void OnSaveButtonClicked(object sender, EventArgs args)
    {
        note.Save(FILENAME);
        loadButton.IsEnabled = true;
    }

    void OnLoadButtonClicked(object sender, EventArgs args)
    {
        note.Load(FILENAME);
    }
}
```

As you test this new version, you might want to restore the phone or simulator to a state where no file has yet been saved. You can do that simply by uninstalling the application from the phone or simulator. That uninstall removes all the data stored along with the application as well.

Version 5. Data binding

Xamarin.Forms is mostly about user interfaces, but user interfaces rarely exist in isolation. A case in point is the application being built in this chapter. This user interface is really a visual representation of data and the means through which that data is manipulated.

To accommodate such scenarios, .NET and Xamarin.Forms provide some built-in facilities for smoothing the links between data and user interfaces. The **NoteTaker5** program incorporates some of these features. In particular, you'll see here how to automate the process of data binding: linking two

properties of two objects so that changes to one property automatically trigger a change to the other.

Streamlining INotifyPropertyChanged classes

Classes that implement `INotifyPropertyChanged` usually have rather more changeable properties than the `Note` class. For this reason, it's a good idea to simplify the `set` accessors, if only to avoid mixing up property and field names, or misspelling the text property name.

One simplification is to encapsulate the actual firing of the event in a protected virtual method:

```
protected virtual void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Now the `Title` property looks like this:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;
            OnPropertyChanged("Title");
        }
    }
    get
    {
        return title;
    }
}
```

The `OnPropertyChanged` method is made protected and virtual because you might write an enhanced `Note` class that derives from this `Note` class but includes more properties. The derived class needs access to this method.

But let's pause a moment to improve out `INotifyPropertyChanged` handling. It's recommended to obtain the handler first, and then perform the check for `null` on and the event call on that same object:

```
protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
}
```

In a multithreaded environment, a PropertyChanged handler might be detached between the null

check and the call, and this code prevents a null-reference exception from occurring.

You can go further in streamlining the `OnPropertyChanged` method. C# 5.0 introduced support for `CallerMemberNameAttribute` and some related attributes. This attribute allows you to replace an optional method argument with the name of the calling method or property.

In the `OnPropertyChanged` method, make the argument optional by assigning `null` to it and precede it with `CallerMemberName` in square brackets:

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

You'll need a `using` directive for `System.Runtime.CompilerServices` for that attribute. Now the `Title` property can call `OnPropertyChanged` with no arguments, and the `propertyName` argument will automatically be set to the property name "Title" because that's where the call to `OnPropertyChanged` is originating:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;
            OnPropertyChanged();
        }
    }
    get
    {
        return title;
    }
}
```

This avoids a potentially misspelled text property name, and allows property names to be changed during program development without worrying about also changing text strings. One of the primary reasons the `CallerMemberName` was invented was to simplify classes that implement `INotifyPropertyChanged`.

It's possible to go even further. You'll need to define a generic method named `SetProperty` (for example) with the `CallerMemberName` attribute, but you'll need to remove it from `OnPropertyChanged`:

```
bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
{
    if (Object.Equals(storage, value))
        return false;

    storage = value;
```

```
    OnPropertyChanged(propertyName);
    return true;
}

protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
}
```

The `SetProperty` method requires access to the backing field and the new value, but automates the rest of the process and returns `true` if the property was changed. (You might need to use this return value if you're doing some additional processing in the `set` accessor.) Now the `Title` property looks like this:

```
public string Title
{
    set
    {
        SetProperty(ref title, value);
    }
    get
    {
        return title;
    }
}
```

Although `SetProperty` is a generic method, the C# compiler can deduce the type from the arguments. The whole property definition has become so short, you can write the accessors concisely on single lines without obscuring the operations:

```
public string Title
{
    set { SetProperty(ref title, value); }
    get { return title; }
}
```

Here is the new `Note` class in the PCL project **NoteTaker5**:

```
class Note : INotifyPropertyChanged
{
    string title, text;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Title
    {
        set { SetProperty(ref title, value); }
        get { return title; }
    }

    public string Text
    {
```

```
        set { SetProperty(ref text, value); }
        get { return text; }
    }

    public void Save(string filename)
    {
        string text = this.Title + "\n" + this.Text;
        FileHelper.WriteAllText(filename, text, () => { });
    }

    public void Load(string filename)
    {
        FileHelper.ReadAllText(filename, (string text) =>
            {
                // Break string into Title and Text.
                int index = text.IndexOf('\n');
                this.Title = text.Substring(0, index);
                this.Text = text.Substring(index + 1);
            });
    }

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This type of streamlining obviously makes much more sense for classes with more than just two properties, but then it begins making lots of sense.

A peek into BindableObject and bindable properties

You've seen how you can define a class that implements the `INotifyPropertyChanged` interface. You'll probably be interested to learn that many of the classes in Xamarin.Forms—including all the view, layout, and page classes—also implement `INotifyPropertyChanged`. All these classes have a `PropertyChanged` event that your applications can use to be notified when properties of these classes change.

For example, the **NoteTaker4** program keeps the `Title` property of the `Note` class updated from the `Text` property of the `Entry` view like so:

```
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };
```

You can do pretty much the same thing by installing a handler for the `PropertyChanged` event of the `Entry` view and checking for the `Text` property:

```
entry.PropertyChanged += (sender, args) =>
    {
        if (args.PropertyName == "Text")
            note.Title = entry.Text;
    };
```

In a sense, the `TextChanged` event defined by `Entry` and `Editor` is redundant and unnecessary. It is provided solely for purposes of programmer convenience.

Many of the classes in Xamarin.Forms implement `INotifyPropertyChanged` automatically because they derive from a class named `BindableObject` that implements this interface. `BindableObject` also defines a protected virtual method named `OnPropertyChanged`.

As its name implies, `BindableObject` is important to the support of data binding in Xamarin.Forms, yet the implementation of `INotifyPropertyChanged` by this class is only part of the story and, to be honest, the easier part.

Let's begin exploring the more arcane part of `BindableObject` with some experimentation. In one of the previous versions of the **NoteTaker** program, try initializing the `Entry` view with some text:

```
entry.Text = "This is some text";
```

Now when you run the program the `Entry` is initialized with this text. But now try replacing that property setting with this method call:

```
entry.SetValue(Entry.TextProperty, "This is some text");
```

This works as well. These two statements are functionally identical.

Look at that weird first argument to `SetValue`: It's something called `Entry.TextProperty`, which indicates that it's static, but it's not a property at all. It's static *field* of the `Entry` class. It's also read-only, and it's defined in the `Entry` class something like this:

```
public static readonly BindableProperty TextProperty;
```

It's a little odd for a field of a class to be named `TextProperty`, but there it is. Because it's static, however, it exists independently of any `Entry` objects that might or might not exist.

If you look in the documentation of the `Entry` class, you'll see that it defines four properties—`Text`, `TextColor`, `IsPassword`, and `Placeholder`—and you'll also see four corresponding public static read-only fields of type `BindableProperty` with the names `TextProperty`, `TextColor-Property`, `IsPasswordProperty`, and `PlaceholderProperty`.

These properties and fields are closely related. Indeed, internal to the `Entry` class, the `Text` property is defined like this:

```
public string Text
{
    set { SetValue(Entry.TextProperty, value); }
    get { return (string)GetValue(Entry.TextProperty); }
}
```

So you see why it is that your application calling `SetValue` on `Entry.TextProperty` is exactly equivalent to setting the `Text` property and perhaps just a tinier bit faster!

The internal definition of the `Text` property in `Entry` isn't secret information. This is standard code. The `SetValue` and `GetValue` methods are defined by `BindableObject`, the same class that implements `INotifyPropertyChanged` for many Xamarin.Forms classes. All the real work involved with maintaining the `Text` property is going on in these `SetValue` and `GetValue` calls. Casting is required for the `GetValue` method because it's defined as returning `object`.

The static `Entry.TextProperty` object is of type `BindableProperty`, which you might correctly surmise is a class related to `BindableObject`, but it's important to keep them distinct in your mind: `BindableObject` is the class from which many Xamarin.Forms classes derive and that provides support for objects of type `BindableProperty`.

The `BindableProperty` objects effectively extend the functionality of standard C# properties. Bindable properties provide systematic ways to:

- Define properties
- Give properties default values
- Store their current values
- Provide mechanisms for validating property values
- Maintain consistency among related properties in a single class
- Respond to property changes
- Trigger notifications when a property is about to change and has changed

In addition, `BindableObject` and `BindableProperty` provide mechanisms for animation and data binding. They are a vital part of the infrastructure of Xamarin.Florms.

The close relationship of a property named `Text` with a `BindableProperty` named `TextProperty` is reflected in the way that programmers speak about these properties: Sometimes a programmer says that the `Text` property is "backed by" a `BindableProperty` named `TextProperty` because `TextProperty` provides infrastructure support for the `Text` property. But a common shortcut is to say that `Text` is itself a "bindable property" and generally no one will be confused.

Not every Xamarin.Forms property is a bindable property. Neither the `Content` property of

ContentPage nor the Children property of Layout<T> (from which StackLayout derives) is a bindable property. Sometimes changes to nonbindable properties result in the PropertyChanged event being fired, but that's not guaranteed. The PropertyChanged event is only guaranteed for bindable properties.

Later in this book you'll see how to define your own bindable properties.

Automated data bindings

As you've seen, it's possible to install event handlers on the Entry and Editor to determine when the Text property changes, and to use that occasion to set the Title and Text properties of the Note class. Similarly, you can use the PropertyChanged event of the Note class to keep the Entry and Editor updated.

In other words, the **NoteTaker4** program has paired up objects and properties so that they track each other's values. As one property changes, the other is updated.

It turns out that tasks like this are very common, and this is why Xamarin.Forms allows such tasks to be automated with a technique called *data binding*.

Data bindings involve a source and a target. The source is the object and property that changes, and the target is the object and property that is changed as a result. But that's a simplification. Although the distinction between target and source is clearly defined in any particular data binding, sometimes the properties affect each other in different ways: Sometimes the target causes the source to be updated, and sometimes the source and target update each other.

The data binding mechanism in Xamarin.Forms uses the PropertyChanged event to determine when a source property has changed. Therefore, a source property used in a data binding must be part of a class that implements INotifyPropertyChanged (either directly or through inheritance), and the class must fire a PropertyChanged event when that property changes. (Actually this is not entirely true: If the source property never changes, a PropertyChanged event is *not* required for the data binding to work, but it's only a "one time" data binding and not very interesting.)

The target property of a data binding must be backed by a BindableProperty. As you'll see, this requirement is imposed by the programming interface for data bindings. You cannot set a data binding without referencing a BindableProperty object!

In the **NoteTaker5** program, the Text properties of Entry and Editor are the binding targets because they are backed by bindable properties named TextProperty. The Title and Text properties of the Note class are the binding sources. The Note class implements INotifyProperty-0Changed so PropertyChanged events are fired when these source properties changes.

You can define a data binding to keep a target property updated with the value of a source property with two statements: The first statement associates the two objects by setting the BindingContext property of the target object to the source object. In this case that's the instance of the Note class:

```
entry.BindingContext = note;
```

The second statement makes use of a `SetBinding` method on the target. These `SetBinding` calls come in several different forms. `BindableObject` itself defines one `SetBinding` method, and the `BindableObjectExtensions` class defines two `SetBinding` extension methods. Here's the simplest:

```
entry.SetBinding(Entry.TextProperty, "Title");
```

You'll see more complex data bindings in the chapters ahead. The target property here is the `Text` property of `Entry`. That's specified in the first argument. The `Title` property—specified here as a text string—is assumed to be a property of whichever object has been defined as the BindingContext of the `Entry` object, which in this case is a `Note` object.

Similarly, you can set a data binding for the `Editor`:

```
editor.BindingContext = note;
editor.SetBinding(Editor.TextProperty, "Text");
```

The first argument of `SetBinding` must be a `BindableProperty` object defined by the target class (`Editor` in this case) or inherited by the target class.

Part of the infrastructure that `BindableProperty` provides is a default binding mode that defines the relationship between the source and the target. The `BindingMode` enumeration has four members:

- `Default`
- `OneWay` — source updates target, the normal case
- `OneWayToSource` — source updates target
- `TwoWay` — source and target update each other

For the `TextProperty` objects defined by `Entry` and `Editor`, the default binding mode is `BindingMode.TwoWay`. This means that these four statements actually define a two-way data binding: Any change to the `Title` property of the `Note` object is reflected in the `Text` property of the `Entry`, and vice versa, and the same goes for the `Editor`.

The **NoteTaker4** program included three event handlers—the `PropertyChanged` handler for the `Note` class and the `TextChanged` handlers for the `Entry` and `Editor`—to keep these pairs of properties in synchronization. Those three event handlers are no longer required if they are replaced by the four statements you've just seen:

```
entry.BindingContext = note;
entry.SetBinding(Entry.TextProperty, "Title");

editor.BindingContext = note;
editor.SetBinding(Editor.TextProperty, "Text");
```

Internally, these four statements result in similar event handlers being set. That's how the data-binding mechanism works.

However, these four statements can actually be reduced to three statements. Here's how:

The `BindingContext` property has a very special characteristic. It is very likely that more than one data binding on a page has the same `BindingContext`. That is true for this little example. For this reason, the `BindingContext` property is propagated through the visual tree of a page. In other words, if you set the `BindingContext` on a page, it will propagate to all the views on that page except for those views that have their own `BindingContext` properties set to something else. You can set `BindingContext` on a `StackLayout` and it will propagate to all the children (and other descendants) of that `StackLayout`. The two `BindingContext` settings shown above can be replaced with one set on the page itself:

```
this.BindingContext = note;
```

These automated data bindings are part of the **NoteTaker5Page** class. Also, the handlers for the **Load** and **Save** buttons have become lambda functions, all the variables have been moved to the constructor, and the code is looking quite sleek at this point:

```
class NoteTaker5Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    public NoteTaker5Page()
    {
        // Create Entry and Editor views.
        Entry entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        Editor editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Set data bindings.
        Note note = new Note();
        this.BindingContext = note;
        entry.SetBinding(Entry.TextProperty, "Title");
        editor.SetBinding(Editor.TextProperty, "Text");

        // Create Save and Load buttons.
        Button saveButton = new Button
        {
            Text = "Save",
            HorizontalOptions = LayoutOptions.CenterAndExpand
```

```
        };

        Button loadButton = new Button
        {
            Text = "Load",
            IsEnabled = false,
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };

        // Set Clicked handlers.
        saveButton.Clicked += (sender, args) =>
            {
                note.Save(FILENAME);
                loadButton.IsEnabled = true;
            };

        loadButton.Clicked += (sender, args) => note.Load(FILENAME);

        // Check if the file is available.
        FileHelper.Exists(FILENAME, (exists) =>
            {
                loadButton.IsEnabled = exists;
            });

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }
}
```

By replacing the explicit event handlers with data bindings, we've also managed to forget all about the problem of the `PropertyChanged` event in the `Note` class being fired from a secondary thread. When you use a data binding, you don't have to worry about that.

Eventually, the final **NoteTaker** application won't be dealing with a single `Note` object. It will have a whole collection of `Note` objects. With these data bindings in place, you can view and edit any one of these `Note` objects by setting it to the `BindingContext` of the page. But that's for the next chapter.

Meanwhile, let's see if we can simplify this page code even more. One prime candidate is the code that enables the **Load** button based on the `FileHelper.Exists` call. Perhaps solving that problem will simplify some other parts of the program as well.

Version 6. Awaiting results

As you've seen, asynchronous operations can be difficult to manage. Sometimes code executes in a different order than you anticipated and some thought is required to figure out what's going on. Asynchronous programming will likely never be quite as simple as single-threaded coding, but some of the difficulty in working with asynchronous operations has been alleviated with C# 5.0, released in 2012. C# 5.0 introduced a revolutionary change in the way that programmers deal with asynchronous operations. This change consists of a keyword named `async` and an operator named `await`.

The `async` keyword is mostly for purposes of backward compatibility. The `await` operator is the big one. It allows programmers to work with asynchronous functions almost as if they were relatively normal imperative programming statements without callback methods.

Let's look at a generalized use of an asynchronous method that might appear in a Windows Phone 8 program. You have a method in your program that calls an asynchronous method in the operating system and sets a `Completed` handler implemented as a separate method. The comments indicate some other code that might appear in the method:

```
void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    asyncOp.Completed = MyCompletedHandler;
    // And perhaps still more code
}
```

The statement that sets the `Completed` property returns quickly while the actual asynchronous operation goes on in the background. All the code in `SomeMethod` will execute before the callback method is called. Here's what the `Completed` handler might look like if you chose to ignore cancellations or errors encountered in the background process:

```
void MyCompletedHandler(IAsyncOperation<SomeType> op, AsyncStatus status)
```

```
{
    SomeType myResult = op.GetResults();
    // Code using the result of the asynchronous operation
}
```

The handler can also be written as a lambda function.

```
void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    asyncOp.Completed = (op, status) =>
    {
        SomeType myResult = op.GetResults();
        // Code using the result of the asynchronous operation
    };
    // And perhaps still more code
}
```

The additional code indicated with the comment "And perhaps still more code" will execute and `SomeMethod` will return before the code in the `Completed` handler executes. The order that the code appears in the method is not the same order that the code executes, and this is one reason why using lambda functions for asynchronous operations can sometimes be a bit confusing.

Here's how `SomeMethod` can be rewritten using the `await` operator:

```
async void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    // And perhaps still more code
    SomeType myResult = await asyncOp;
    // Code using the result of the asynchronous operation
}
```

Notice that `SomeMethod` now includes the `async` modifier. This is required for backward compatibility. In versions of C# prior to 5.0, `await` was not a keyword so it could be used as a variable name. To prevent C# 5.0 from breaking that code, the `async` modifier is required to indicate a method that includes `await`.

The `Completed` handler still exists in this code, but it's not exactly obvious where it is. It's basically everything to the left of the `await` operator, and everything below the statement containing the `await` operator.

The C# compiler performs the magic. The compiler recognizes `IAsyncOperation` as encapsulating an asynchronous method call and basically turns `SomeMethod` into a state machine. The method executes normally up until the `await` operator, and then the method returns. At this point, the background process is running and other code in the program can run as well. When the background process is completed, execution of the method resumes with the assignment to the `myResult`

variable.

If you can organize your code to eliminate the two comments labeled as "Some additional code" and "And perhaps still more code" you can skip the assignment to the `IAsyncOperation` object and just get the result:

```
async void SomeMethod()
{
    // Often some initialization code
    SomeType myResult = await sysClass.LongJobAsync(…);
    // Code using the result of the asynchronous operation
}
```

This is how `await` is customarily used—as an operator between a method that runs asynchronously and the assignment statement to save the result.

Don't let the name of the `await` operator fool you! The code doesn't actually sit there waiting. The `SomeMethod` method returns and the processor is free to run other code. Only when the asynchronous operation has completed does the code in `SomeMethod` resume execution where it left off.

Of course, any time we're dealing with file I/O, we should be ready for problems. What happens if the file isn't there, or you run out of storage space? Even when problems don't occur, sometimes error results are normal: The Windows Phone version of the `Exists` method uses an error reported in the `Completed` handler to determine if the file exists or not. So how are errors handled with `await`?

If you use `await` with an asynchronous operation that encounters an error or is cancelled, `await` throws an exception. If you need to handle errors or cancellations, you can put the `await` operator in a `try` and `catch` block, looking something like this:

```
SomeType myresult = null;
try
{
    myresult = await sysClass.LongJobAsync(…);
}
catch (OperationCanceledException)
{
    // handle a cancellation
}
catch (Exception exc)
{
    // handle an error
}
```

Now let's use `await` in some real code. Here's the Windows Phone version of the old `ReadAllText` method in **NoteTaker5**:

```
public void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
    createOp.Completed = (asyncInfo1, asyncStatus1) =>
    {
```

```
        IStorageFile storageFile = asyncInfo1.GetResults();
        IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                storageFile.OpenReadAsync();
        openOp.Completed = (asyncInfo2, asyncStatus2) =>
        {
            IRandomAccessStream stream = asyncInfo2.GetResults();
            DataReader dataReader = new DataReader(stream);
            uint length = (uint)stream.Size;
            DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
            loadOp.Completed = (asyncInfo3, asyncStatus3) =>
            {
                string text = dataReader.ReadString(length);
                dataReader.Dispose();
                completed(text);
            };
        };
    };
}
```

This has three nested `Completed` handlers. Because the method returns before even the first `Completed` handler executes, it is not possible to return the contents of the file from the method. The file contents must instead be returned through a function passed to the method.

Here's how it can be rewritten using `await`:

```
public async void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    IRandomAccessStream stream = await storageFile.OpenReadAsync();
    DataReader dataReader = new DataReader(stream);
    uint length = (uint)stream.Size;
    await dataReader.LoadAsync(length);
    string text = dataReader.ReadString(length);
    dataReader.Dispose();
    completed(text);
}
```

Notice the `async` modifier on the method. The `async` modifier does not change the signature of the method, so this method is still considered a proper implementation of the `ReadAllText` method in the `IFileHelper` interface.

The `await` operator appears three times; the first two times on methods that return an object and the third time on the `LoadAsync` method that just performs an operation without returning anything.

Behind the scenes, the C# compiler divides this method into four chunks of execution. The method returns to the caller (the `Note` class) at the first `await` operator and then resumes when the `GetFileAsync` method has completed, leaving again at the next `await`, and so on.

Actually, it's possible for the `ReadAllText` method to execute sequentially from start to finish. If the asynchronous methods complete their operations before the `await` operator is evaluated, execution just continues as if it were a normal function call. This is a performance optimization that

often plays a role in the relatively fast solid state file I/O on mobile devices.

Let's improve this `ReadAllText` method a bit. The `DataReader` class implements the `IDisposable` interface and includes a `Dispose` method. Failing to call `Dispose` can leave the file open. Calling `Dispose` also closes the `IRandomAccessStream` on which the `DataReader` is based. `IRandomAccessStream` also implements `IDisposable`.

It's a good idea to enclose `IDisposable` objects in `using` blocks. This ensures that the `Dispose` method is automatically called even if an exception is thrown inside the block. A better implementation of `ReadAllText` is this:

```
public async void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
    {
        using (DataReader dataReader = new DataReader(stream))
        {
            uint length = (uint)stream.Size;
            await dataReader.LoadAsync(length);
            string text = dataReader.ReadString(length);
            completed(text);
        }
    }
}
```

Now the explicit call to `Dispose` is not required.

Of course, we still have the annoyance of passing a callback function to the `ReadAllText` method. But what's the alternative? If the method actually returns to the caller at the first `await` operator, how can the method return the text contents of the file? Trust the C# compiler. If it can implement `await`, surely it can also allow you to create your own asynchronous methods.

Let's replace `ReadAllText` with a method named `ReadTextAsync`. The new name reflects the fact that this method is itself asynchronous and can be called with the `await` operator to return a `string` with the contents of the file.

To do this, the `ReadTextAsync` method needs to return a `Task` object. The `Task` class is defined in `System.Threading.Tasks` and is the standard .NET representation of an asynchronous operation. The `Task` class is quite extensive, but only a little bit of it is necessary in this context. The `System.Threading.Tasks` namespace actually defines two `Task` classes:

- `Task` for asynchronous methods that return nothing
- `Task<TResult>` for asynchronous methods that return an object of type `TResult`.

These are the .NET equivalences of the Windows 8 `IAsyncAction` and `IAsyncOperation<TResult>` interfaces, and there are extension methods that convert the `Task` objects to `IAsyncAction` and `IAsyncOperation<TResult>` objects.

Instead of returning `void`, this new method can return `Task<string>`. The callback argument isn't required, and inside the method, the file's contents are returned as if this were a normal method:

```
public async Task<string> ReadTextAsync(string filename)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
    {
        using (DataReader dataReader = new DataReader(stream))
        {
            uint length = (uint)stream.Size;
            await dataReader.LoadAsync(length);
            return dataReader.ReadString(length);
        }
    }
}
```

The compiler handles the rest. The `return` statement seems to return a `string` object, which is the return value of the `ReadString` method, but the C# compiler automatically wraps that value in a `Task<string>` object actually returned from the method at the execution of the first `await` operator.

This `ReadTextAsync` method can now be called using an `await` operator.

Of course, redefining the method signature of these file I/O functions has an impact throughout the program. If we want to continue to use `DependencyService` to call platform-independent methods—and that is highly desirable—the iOS and Android methods should have the same signature. This means that the **NoteTaker6** version of `IFileHelper` consists of these three asynchronous methods:

```
using System.Threading.Tasks;

namespace NoteTaker6
{
    public interface IFileHelper
    {
        Task<bool> ExistsAsync(string filename);

        Task WriteTextAsync(string filename, string text);

        Task<string> ReadTextAsync(string filename);
    }
}
```

There are no longer any callback functions in the methods. The `Task` object has its own callback mechanism.

Here's the complete Windows Phone version of the `FileHelper` implementation of `IFileHelper`:

```
using System;
```

```
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Streams;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker6.WinPhone.FileHelper))]

namespace NoteTaker6.WinPhone
{
    class FileHelper : IFileHelper
    {
        public async Task<bool> ExistsAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;

            try
            {
                await localFolder.GetFileAsync(filename);
            }
            catch
            {
                return false;
            }
            return true;
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile =
                    await localFolder.CreateFileAsync(filename,
                                CreationCollisionOption.ReplaceExisting);

            using (IRandomAccessStream stream =
                    await storageFile.OpenAsync(FileAccessMode.ReadWrite))
            {
                using (DataWriter dataWriter = new DataWriter(stream))
                {
                    dataWriter.WriteString(text);
                    await dataWriter.StoreAsync();
                }
            }
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.GetFileAsync(filename);
            using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
            {
                using (DataReader dataReader = new DataReader(stream))
                {
                    uint length = (uint)stream.Size;
                    await dataReader.LoadAsync(length);
```

```
                return dataReader.ReadString(length);
            }
        }
    }
}
```

Notice the use of the `try` and `catch` block on the `await` operation in the `ExistsAsync` method.

The `WriteTextAsync` method doesn't return a value, and the return value of the method is simply `Task`. Such a method doesn't require an explicit `return` statement. A method that returns `Task<TResult>` needs a `return` statement with a `TResult` object. In either case, all the asynchronous calls in the method should be preceded with `await`. (There are alternatives but they are somewhat more complicated. Asynchronous methods without any `await` operators can also be handled somewhat differently as you'll see shortly.)

For the iOS and Android versions, the methods now need to return `Task` and `Task<TResult>` objects, but up to this point the methods themselves haven't been asynchronous. One solution is to switch to using asynchronous file I/O, at least in part. Many of the I/O methods in `System.IO` have asynchronous versions. That's what's been done here with the `WriteTextAsync` and `ReadTextAsync` methods:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker6.iOS.FileHelper))]

namespace NoteTaker6.iOS
{
    class FileHelper : IFileHelper
    {
        public Task<bool> ExistsAsync(string filename)
        {
            string filepath = GetFilePath(filename);
            bool exists = File.Exists(filepath);
            return Task<bool>.FromResult(exists);
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            using (StreamWriter writer = File.CreateText(filepath))
            {
                await writer.WriteAsync(text);
            }
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            string filepath = GetFilePath(filename);
```

```
            using (StreamReader reader = File.OpenText(filepath))
            {
                return await reader.ReadToEndAsync();
            }
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

However, there is no asynchronous version of the `File.Exists` method, so a `Task<bool>` is simply constructed from the result using the static `FromResult` method.

The Android implementation of `FileHelper` is the same as the iOS version. The PCL project in **NoteTaker6** has a new static `FileHelper` method that caches the `IFileHelper` object and hides all the `DependencyService` calls:

```
using System.Threading.Tasks;
using Xamarin.Forms;

namespace NoteTaker6
{
    static class FileHelper
    {
        static IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public static Task<bool> ExistsAsync(string filename)
        {
            return fileHelper.ExistsAsync(filename);
        }

        public static Task WriteTextAsync(string filename, string text)
        {
            return fileHelper.WriteTextAsync(filename, text);
        }

        public static Task<string> ReadTextAsync(string filename)
        {
            return fileHelper.ReadTextAsync(filename);
        }
    }
}
```

These methods simply return the same return values as the underlying asynchronous methods.

The `Note` class is mostly the same as before, but the `Save` and `Load` methods are now `SaveAsync` and `LoadAsync`:

```
using System;
using System.ComponentModel;
```

```
using System.Runtime.CompilerServices;
using System.Threading.Tasks;

namespace NoteTaker6
{
    class Note : INotifyPropertyChanged
    {
        …

        public Task SaveAsync(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            return FileHelper.WriteTextAsync(filename, text);
        }

        public async Task LoadAsync(string filename)
        {
            string text = await FileHelper.ReadTextAsync(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }

        …
    }
}
```

Neither of these methods returns a value, but because they are asynchronous, the methods return a `Task` object that can be awaited. There are a couple ways to deal with such methods. The `SaveAsync` method simply returns the return value from `FileHelper.WriteTextAsync`, which is also a `Task` object. The `LoadAsync` method has no `return` statement, although it could surely end with an empty `return` statement. The `SaveAsync` method could have an implicit empty `return` statement but the `WriteTextAsync` call would have to be preceded with `await`, and because of the existence of that `await` operator, the method would need an `async` modifier:

```
public async Task SaveAsync(string filename)
{
    string text = this.Title + "\n" + this.Text;
    await FileHelper.WriteTextAsync(filename, text);
}
```

These new function definitions require changes to the code in the `NoteTaker6Page` constructor as well, and you have a couple choices. You can define the `Clicked` handler for the **Load** button like so:

```
loadButton.Clicked += (sender, args) =>
    {
        note.LoadAsync(FILENAME);
    };
```

You'll get a warning from the compiler that you might consider using the `await` operator. But strictly speaking, you don't need to. What happens is that the `Clicked` handler calls `LoadAsync` but doesn't wait for it to complete. The `Clicked` handler returns back to the button that fired the event before the file has been loaded.

You can use `await` in a lambda function but you must precede the argument list with the `async` modifier:

```
loadButton.Clicked += async (sender, args) =>
    {
        await note.LoadAsync(FILENAME);
    };
```

For the **Save** button, you probably don't want to enable the **Load** button until the save operation has completed, so you'll want the `await` in there:

```
saveButton.Clicked += async (sender, args) =>
    {
        await note.SaveAsync(FILENAME);
        loadButton.IsEnabled = true;
    };
```

Actually, if you start thinking about asynchronous file I/O, you might start getting nervous—and justifiably so. For example, what if you press the **Save** button and, while the file is still in the process of being saved, you press the **Load** button? Will an exception result? Will you only load half the file because the other half hasn't been saved yet?

It's possible to avoid such problems on the user-interface level. If you want to prohibit a button press at a particular time, you can disable the button. Here's how the program can prevent the buttons from interfering with each other or with themselves:

```
saveButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.SaveAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };

loadButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.LoadAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };
```

It's unlikely that you'll run into problems with these very small files and solid-state memory, but it never hurts to be too careful.

As you'll recall, the **Load** button must be initially disabled if the file doesn't exist. The `await` operator is a full-fledged C# operator, so you should be able to do something like this:

```
Button loadButton = new Button
{
    Text = "Load",
    IsEnabled = await FileHelper.ExistsAsync(FILENAME),
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
```

Yes, this works!

And yet, it doesn't. The problem is that the method that contains this code must have an `async` modifier and the `async` modifier is not allowed on a constructor. But there's a way around this restriction. You can put all the initialization code in a method named `Initialize` with the `async` modifier and then call it from the constructor:

```
public NoteTaker6Page()
{
    Initialize();
}

async void Initialize()
{
    …
}
```

The `Initialize` method will execute up to the point of the `await` operator and then return back to the constructor, which will then return back to the code that instantiated the class. The remainder of the `Initialize` method continues after the `ExistsAsync` method returns a value and the `IsEnabled` property is set.

But in the general case, do you want a good chunk of your page initialization to be delayed until a single property is set that has no other effect on the page? Probably not.

Even with the availability of `await`, there are times when it makes sense to devote a completed handler to a chore. When an asynchronous method returns a `Task` object, the syntax is a little different for specifying a completed handler, but here it is:

```
FileHelper.ExistsAsync(FILENAME).ContinueWith((task) =>
    {
        loadButton.IsEnabled = task.Result;
    });
```

Here's the complete `NoteTaker6Page` class:

```
class NoteTaker6Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    public NoteTaker6Page()
    {
```

```
// Create Entry and Editor views.
Entry entry = new Entry
{
    Placeholder = "Title (optional)"
};

Editor editor = new Editor
{
    Keyboard = Keyboard.Create(KeyboardFlags.All),
    BackgroundColor = Device.OnPlatform(Color.Default,
                                        Color.Default,
                                        Color.White),
    VerticalOptions = LayoutOptions.FillAndExpand
};

// Set data bindings.
Note note = new Note();
this.BindingContext = note;
entry.SetBinding(Entry.TextProperty, "Title");
editor.SetBinding(Editor.TextProperty, "Text");

// Create Save and Load buttons.
Button saveButton = new Button
{
    Text = "Save",
    HorizontalOptions = LayoutOptions.CenterAndExpand
};

Button loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};

// Set Clicked handlers.
saveButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.SaveAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };

loadButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.LoadAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };
```

```
        // Check if the file is available.
        FileHelper.ExistsAsync(FILENAME).ContinueWith((task) =>
            {
                loadButton.IsEnabled = task.Result;
            });

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }
}
```

With no error handling, the program is implicitly assuming that there will be no problems encountered when loading and saving files. Error handling can be implemented by enclosing any asynchronous method call with `await` in a `try` and `catch` block. If you want to deal with errors "silently" without informing the user, you can implement this check in the `Note` class. If you need to display an alert to the user, it makes more sense to perform the check in the `ContentPage` derivative.

What's next?

Much of the infrastructure is in place for storing and retrieving a single note. We are now ready to save and retrieve multiple notes and display them in a scrollable list for reference or editing.

CHAPTER 4

Building an app: Architecture

The **NoteTaker** application has evolved from a concept to a single-page program that contains much of the basic infrastructure to save and retrieve a note. It must now be expended to a multipage application that can display and maintain multiple notes.

This chapter illustrates the steps to implement all the necessary features for that enhanced version. Two important Xamarin.Forms visual elements are necessary: The first is `NavigationPage`, which allows a program to navigate among multiple pages, and the second is `ListView`, which displays a scrollable and selectable list of items, usually of the same type. In this program, `ListView` will display multiple `Note` objects.

Also required for specifying some program actions is `ToolbarItem`. Strictly speaking, `ToolbarItem` is not a visual element, but it's used to specify items for a toolbar displayed at the top of the page (on iOS and Android) or the bottom (on Windows Phone).

From the programmer's perspective, one of the big differences between programming for traditional desktop applications and programming for mobile applications is the need to understand the application lifecycle. Between the time that a mobile app is launched and the time it is terminated, the app might be suspended. It is important at this time for applications to save application data—including transient data—necessary to re-create the contents of the application when it's restarted. Application lifecycle programming is also a major part of creating a real app.

Version 7. Page navigation

So far, all the programs in this book have been based around a single instance of a single class that derives from `ContentPage`. In contrast, **NoteTaker7** contains two classes that derive from `ContentPage`, named `HomePage` and `NotePage`. When the program starts up, it displays `HomePage`, which lists all the notes you've entered in previous sessions. From `HomePage` the user is able to navigate to `NotePage` for entering new notes, or for editing or deleting existing notes, and then navigate back to `HomePage`, which displays an updated list.

The methods that Xamarin.Forms makes available for page navigation are all defined by the `INavigation` interface in the `Xamarin.Forms` namespace:

```
public interface INavigation
{
    Task PushAsync(Page page);
    Task<Page> PopAsync();
    Task PopToRootAsync();
    Task PushModalAsync(Page page);
```

```
    Task<Page> PopModalAsync();
}
```

These are all asynchronous methods, which means they return immediately while the navigation operation proceeds.

As you can probably deduce from the names of these methods, page navigation in Xamarin.Forms works like a familiar last-in-first-out stack: Each `PushAsync` call pushes the current page on the stack and navigates to the specified page. Each call to `PopAsync` pops the previous page from the stack and navigates back to that popped page. This architecture is very familiar to users because web browsers work similarly.

Explicit calls to `PopAsync` are not always required because the user can often navigate backward using the hardware back button on the Android or Windows Phone, or a back button automatically supplied as part of the page navigation user interface by iOS and Android. The `PopToRootAsync` call essentially clears the stack and goes back to the home page. A future chapter (not in this Preview Edition) will contain a more extensive discussion of pages and navigation, including the `Modal` methods for popup page navigation.

The `INavigation` interface is implemented by a class internal to Xamarin.Forms but which is accessible through the object returned from the `Navigation` property defined by `VisualElement`. Very often it's convenient to use the `Navigation` property of the page itself to navigate to another page. For example:

```
this.Navigation.PushAsync(new AnotherPage());
```

However, you can use the `Navigation` property of any view on the page if that's more convenient.

To enable page navigation within a Xamarin.Forms application, the `App.GetMainPage` method should return an instance of `NavigationPage` rather than `ContentPage`. Fortunately, this change is exceptionally easy. If `HomePage` is derived from `ContentPage` (which is the case in the **NoteTaker7** application), then `App.GetMainPage` simply passes a new instance of that page to the `NavigationPage` constructor and returns the result. Here's the `App` class in **NoteTaker7**:

```
namespace NoteTaker7
{
    public class App
    {
        public static Page GetMainPage()
        {
            return new NavigationPage(new HomePage());
        }
    }
}
```

That turns a `ContentPage` object into a `NavigationPage` object.

Strictly speaking, this wrapping of a `ContentPage` in a `NavigationPage` is not required for Windows Phone, but you need it for iOS and Android, so it's easiest to use the same code for all three

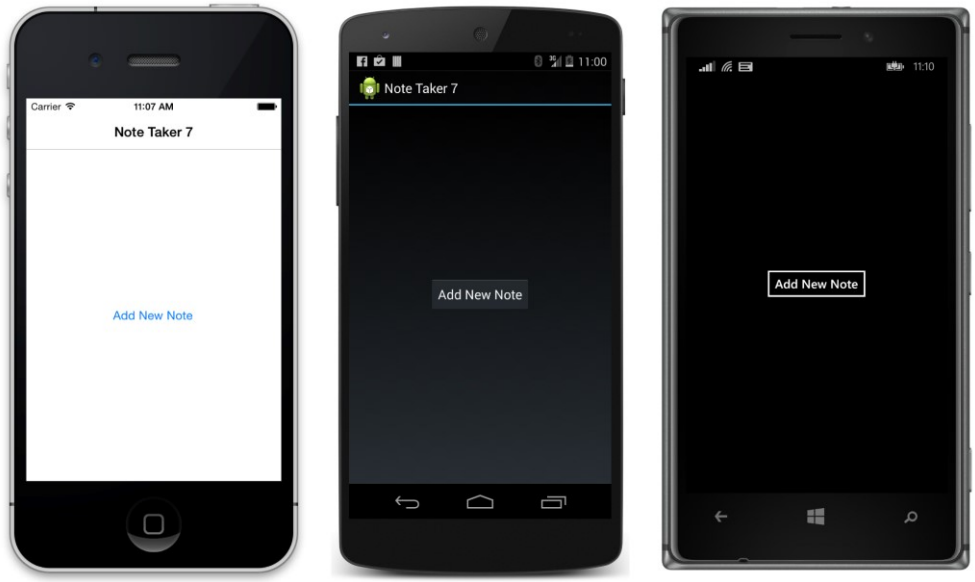
platforms.

For iOS and Android, this change has a demonstrable effect on the visual appearance of the page. An area at the top of the page is reserved for a page title that you can set using the `Title` property that `Page` defines. You should set this `Title` property whenever you've enabled page navigation.

The `HomePage` class in **NoteTaker7** derives from `ContentPage`, and the constructor sets that `Title` property. The constructor also creates a centered `Button` with the text "Add New Note" and a `Clicked` handler that fabricates a unique filename based on the current date and time and navigates to `NotePage`:

```
namespace NoteTaker7
{
    class HomePage : ContentPage
    {
        public HomePage()
        {
            this.Title = "Note Taker 7";

            Button button = new Button
            {
                Text = "Add New Note",
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.Center
            };

            button.Clicked += (sender, args) =>
                {
                    // Create unique filename.
                    DateTime datetime = DateTime.UtcNow;
                    string filename = datetime.ToString("yyyyMMddHHmmssfff") + ".note";

                    // Navigate to new NotePage.
                    this.Navigation.PushAsync(new NotePage(filename));
                };

            this.Content = button;
        }
    }
}
```

Here's how this page looks when **NoteTaker7** is running on the three platforms:



Notice the title on the iPhone and Android. When using `NavigationPage` with the iPhone, you no longer need worry about anything on the page occupying the same space as the status bar either on the home page or on any page navigated to from this page.

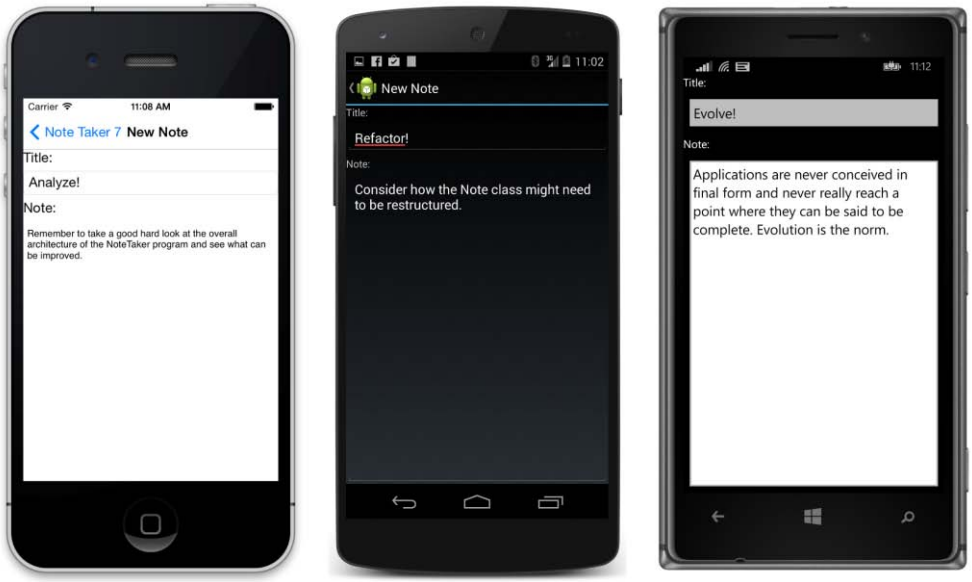
The `Note`, `IFileHelper`, and `FileHelper` classes in **NoteTaker7** are the same as in **NoteTaker6**.

The `NotePage` class is similar to the page classes in previous versions of the program. It derives from `ContentPage` but defines a constructor with a single string parameter indicating the filename of the note. This constructor saves the filename as a private field, sets the `Title` property, and builds the remainder of the page:

```
namespace NoteTaker7
{
    class NotePage : ContentPage
    {
        string filename;
        Note note = new Note();

        public NotePage(string filename)
        {
            this.filename = filename;
            Title = "New Note";

            // Create Entry and Editor views.
            Entry entry = new Entry
            {
                Placeholder = "Title (optional)"
            };
```

```
            Editor editor = new Editor
            {
                Keyboard = Keyboard.Create(KeyboardFlags.All),
                BackgroundColor = Device.OnPlatform(Color.Default,
                                                    Color.Default,
                                                    Color.White),
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Set data bindings.
            this.BindingContext = note;
            entry.SetBinding(Entry.TextProperty, "Title");
            editor.SetBinding(Editor.TextProperty, "Text");

            // Asssemble page.
            this.Content = new StackLayout
            {
                Children =
                {
                    new Label { Text = "Title:" },
                    entry,
                    new Label { Text = "Note:" },
                    editor,
                }
            };
        }

        …

    }
}
```

Here it is:



Again, notice that the `Title` property of this page is displayed at the top of the iOS and Android screens and that a left angle bracket suggests that a tap up there serves to navigate back to the home page. In addition, the leftmost buttons at the bottom of the Android and Windows Phone devices also navigate back to the home page.

But where are the **Load** and **Save** buttons? Well, the **Load** button wouldn't make any sense, because in this version of the program `NotePage` is only displayed with a new filename. The file does not yet exist. However, `NotePage` should have some facility to save the note.

And it does. When working with page navigation, it's often convenient to make use of two protected virtual methods defined by the `Page` class called `OnAppearing` and `OnDisappearing`. The `OnAppearing` method is called as the page is navigated to, and `OnDisappearing` method is called as the page is being navigated from. In the `NotePage` class, `OnAppearing` doesn't do much here (it's shown just for the sake of symmetry), but the `OnDisappearing` override is the perfect opportunity to store the `Note` object:

```
namespace NoteTaker7
{
    class NotePage : ContentPage
    {

        …

        protected override void OnAppearing()
        {
            base.OnAppearing();
        }

        protected override void OnDisappearing()
```

```
        {
            // Only save it if there's some text somewhere.
            if (!String.IsNullOrWhiteSpace(note.Title) ||
                !String.IsNullOrWhiteSpace(note.Text))
            {
                note.SaveAsync(filename);
            }
            base.OnDisappearing();
        }
    }
}
```

This means that the user doesn't need to bother about explicitly saving the note. It's done automatically while navigating back to the home page.

You might wonder: What happens if the user terminates the app without navigating back? Or leaves the app running with the `NotePage` still active and then shuts down the phone? Yes, that's a problem. But it's really an issue involving a subject known as *application lifecycle*, and that will be confronted before this chapter is concluded.

Version 8. Notes in the ListView

As the **NoteTaker** program is enhanced to display multiple notes, a little refactoring is probably in order. Currently, the `Note` object and the filename associated with that `Note` object are not tightly linked. It's possible for a program to call the `SaveAsync` method of `Note` with a different filename than used with the `LoadAsync` method.

Let's fix that problem. In **NoteTaker8**, let's pass a filename to the `Note` constructor, which saves it as a public read-only property named `Filename`. (You'll eventually see the rationale behind making this property public. However, because this `Filename` property is set in the constructor and never changes, there's no reason for the property to fire the `PropertyChanged` event.) That allows the `LoadAsync` and `SaveAsync` methods to be redefined with no arguments. Here are the pertinent parts of the new version:

```
namespace NoteTaker8
{
    class Note : INotifyPropertyChanged
    {
        string title, text;

        public event PropertyChangedEventHandler PropertyChanged;

        public Note(string filename)
        {
            this.Filename = filename;
        }

        public string Filename { private set; get; }
```

```
        …

        public Task SaveAsync()
        {
            string text = this.Title + "\n" + this.Text;
            return FileHelper.WriteTextAsync(this.Filename, text);
        }

        public async Task LoadAsync()
        {
            string text = await FileHelper.ReadTextAsync(this.Filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }

        …
    }
}
```

This means that the new version of `NotePage` must be passed a `Note` object in its constructor rather than the filename:

```
namespace NoteTaker8
{
    class NotePage : ContentPage
    {
        Note note;

        public NotePage(Note note)
        {
            this.note = note;
            Title = "New Note";

            // Create Entry and Editor views.
            Entry entry = new Entry
            {
                Placeholder = "Title (optional)"
            };

            Editor editor = new Editor
            {
                Keyboard = Keyboard.Create(KeyboardFlags.All),
                BackgroundColor = Device.OnPlatform(Color.Default,
                                                    Color.Default,
                                                    Color.White),
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Set data bindings.
            this.BindingContext = note;
```

```
            entry.SetBinding(Entry.TextProperty, "Title");
            editor.SetBinding(Editor.TextProperty, "Text");

            // Assemble page.
            this.Content = new StackLayout
            {
                Children =
                {
                    new Label { Text = "Title:" },
                    entry,
                    new Label { Text = "Note:" },
                    editor,
                }
            };
        }

        …

    }
}
```

The constructor sets the `BindingContext` of itself to this `Note` object.

Enumerating files

The previous version of the program, **NoteTaker7**, saves a new note in local application storage when the user navigates from the `NotePage` and back to the `HomePage`. But the big change in **NoteTaker8** is a facility to display all the notes that have been saved. There are a couple of strategies that might be considered to implement such a feature. It's possible to save a special file in local storage that contains a master list of all the individual filenames of the notes. Or, the program could use file I/O facilities to enumerate all the note files in the folder.

NoteTaker8 implements that second strategy, but this requires that the file helpers be enhanced to include methods to obtain all the files in the application storage folder. Here's the new `IFileHelper` with a new method named `GetFilesAsync`:

```
namespace PlatformHelpers
{
    public interface IFileHelper
    {
        Task<bool> ExistsAsync(string filename);

        Task WriteTextAsync(string filename, string text);

        Task<string> ReadTextAsync(string filename);

        Task<IEnumerable<string>> GetFilesAsync();

        Task DeleteFileAsync(string filename);
    }
}
```

Notice that a `DeleteFileAsync` method has also been added. This is not necessary yet but will be required in the next version of the application.

Also notice that the namespace has been changed to something more generic. It is very likely that the interfaces and classes you create to use with `DependencyService` will also be valuable in other applications. Using a consistent namespace name for all your `DependencyService` helpers will be convenient.

The iOS and Android implementations of the new `IFileHelper` are similar. Here's the iOS version but showing only the new methods:

```
namespace PlatformHelpers.iOS
{
    class FileHelper : IFileHelper
    {

        …

        public Task<IEnumerable<string>> GetFilesAsync()
        {
            // Sort the filenames.
            IEnumerable<string> filenames =
                from filepath in Directory.EnumerateFiles(GetDocsFolder())
                select Path.GetFileName(filepath);

            return Task<IEnumerable<string>>.FromResult(filenames);
        }

        public Task DeleteFileAsync(string filename)
        {
            File.Delete(GetFilePath(filename));
            return Task.FromResult(true);
        }

        string GetDocsFolder()
        {
            return Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        }

        string GetFilePath(string filename)
        {
            return Path.Combine(GetDocsFolder(), filename);
        }
    }
}
```

The `Directory.EnumerableFiles` method returns the fully-qualified filenames, so `GetFilesAsync` uses a little LINQ to strip off the directory path so that only the filenames remain. This is for compatibility with the `WriteTextAsync` and `ReadTextAsync` methods, which require only a filename and provide the directory path themselves.

The Windows Phone version uses a WinRT `StorageFolder` method:

```
namespace PlatformHelpers.WinPhone
{

        …

        public async Task<IEnumerable<string>> GetFilesAsync()
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;

            IEnumerable<string> filenames =
                from storageFile in await localFolder.GetFilesAsync()
                select storageFile.Name;

            return filenames;
        }

        public async Task DeleteFileAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            StorageFile storageFile = await localFolder.GetFileAsync(filename);
            await storageFile.DeleteAsync();
        }
    }
}
```

Finally, the static `FileHelper` class in the Portable Class Library has also been expanded:

```
namespace PlatformHelpers
{
    static class FileHelper
    {
        static IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public static Task<bool> ExistsAsync(string filename)
        {
            return fileHelper.ExistsAsync(filename);
        }

        public static Task WriteTextAsync(string filename, string text)
        {
            return fileHelper.WriteTextAsync(filename, text);
        }

        public static Task<string> ReadTextAsync(string filename)
        {
            return fileHelper.ReadTextAsync(filename);
        }

        public static Task<IEnumerable<string>> GetFilesAsync()
        {
            return fileHelper.GetFilesAsync();
        }

        public static Task DeleteFileAsync(string filename)
        {
```

```
            return fileHelper.DeleteFileAsync(filename);
        }
    }
}
```

These are now the final versions of the `FileHelper` classes (at least for the **NoteTaker** application).

The ListView

NoteTaker8 needs a way to list all the notes. It could use a `Label` view to display the title of each note and then collect all those `Label` views in a `StackLayout` inside a `ScrollView`. But there exists a view that does most of this work for you, and then some. This is `ListView`, and whenever you need to display a scrollable list of objects, this is your go-to view—particularly if the objects are of the same type.

Basically you fill the `ListView` with a collection of objects, and `ListView` displays them. You might think the objects displayed by the `ListView` must be views of some sort, but that is not so. The objects displayed by a `ListView` are generally data objects, and you can control precisely how the `ListView` displays these data objects. (A full discussion of how this works awaits us in a chapter that is unfortunately not in this Preview Edition.)

Items displayed by the `ListView` are selectable. The user simply taps an item to select it. The `ItemSelected` event signals when a new item is selected, and the `SelectedItem` property reveals that item. This facility will be useful in the next version of **NoteTaker** for editing and deleting notes.

You specify the items that are displayed by the `ListView` by setting its `ItemsSource` property to a collection object that implements the `IEnumerable` interface. This is the .NET interface associated with the most generalized type of collection. Arrays implement `IEnumerable`, so an array qualifies. More commonly, the `ItemsSource` property of `ListView` is set to an instance of `List<T>`. For example, if `list` is an instance of `List<T>`, then you can set `ItemsSource` like so:

```
listView.ItemsSource = list;
```

The `ListView` will then display all the items in the collection, with scrolling if necessary.

Very often the collection of data items you want the `ListView` to display is exposed as a property of some other object. For example, suppose a class named `Data` defines a property named `CollectionProp` that implements `IEnumerable`, and `data` is an instance of that class. You can set the `ItemsSource` property like so:

```
listView.ItemsSource = data.CollectionProp;
```

But what happens if the collection changes after the `CollectionProp` property has been set to the `ItemsSource` property?

And what exactly is meant by the word “change” for a collection?

There are actually three different ways that the collection might change! Understanding the different types of change associated with collections will help you design your data and user interface so that the `ListView` always displays the latest and greatest data items.

Here's the first type of change: Perhaps the `CollectionProp` property of the `Data` instance returns a particular `List<T>` object when the application starts running, and then something happens and the property returns a different `List<T>` object. It's a completely different collection with different items. If that is the case, then the `Data` class should implement `INotifyPropertyChanged`, and fire a `PropertyChange` event when the `CollectionProp` property changes. Accordingly, the `ItemsSource` property of `ListView` should not simply be set to the `CollectionProp` property but must be bound to the property with a data binding:

```
listView.BindingContext = data;
listView.SetBinding(ListView.ItemsSourceProperty, "CollectionProp");
```

That's one way the collection might change. Here's a second way: The `List<T>` object returned by the `CollectionProp` property might remain the same while the program is running, but items might be added to or removed from the collection dynamically, or perhaps reordered or resorted in some way, and of course you would like the `ListView` display to reflect the updated contents.

This is the type of change that **NoteTaker8** will need to handle. The `ListView` in **NoteTaker8** will display the collection of notes, but the user can add notes and delete notes to the collection while the program is running. The `ListView` must respond to those changes. At first, this sounds like a difficult requirement. How can the `ListView` determine that the collection that is set to (or bound to) its `ItemsSource` property has acquired a new item or lost an existing item?

Through an event, of course! You already know about the `INotifyPropertyChanged` interface in the `System.ComponentModel` namespace defined like so:

```
public interface INotifyPropertyChanged
{
    event PropertyChangingEventHandler PropertyChanged;
}
```

The `System.Collections.Specialized` namespace contains a related and similar interface named `INotifyCollectionChanged`:

```
public interface INotifyCollectionChanged
{
    event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

A collection class can implement this interface and notify the outside world when an object has been added to or removed from the collection, or the items have been reordered in some way. The event argument associated with `NotifyCollectionChangedEventHandler` provides details on this change.

`ListView` knows about `INotifyCollectionChanged`. Whenever a collection object is set to its

ItemsSource property, ListView checks if that collection object implements this interface. If so, ListView attaches a handler on the CollectionChanged event of that collection object. The ListView is then notified when objects are added to or removed from the collection (or if the objects are reordered in some way), and updates the ListView display accordingly.

Now all we need is an actual class that implements INotifyCollectionChanged, and there is one: It's in the System.Collections.ObjectModel namespace and it's a generic class called ObservableCollection, which is a great name because it implies that by using the CollectionChanged event you can observe what's going on inside the collection.

There is also a third way that a collection can change. Perhaps the collection object itself doesn't change, and objects are not added to, or removed from, the collection, but properties of the objects in the collection change. For example, suppose your collection contains a bunch of Weather objects for major cities. The collection itself never changes but the Weather class contains a Temperature property that changes throughout the day. If this Weather object implements INotifyPropertyChanged and fires PropertyChanged events when the Temperature property changes, the ListView can respond to those changes as well.

NoteTaker8 definitely requires an ObservableCollection, so let's handle that first. Let's create a class that will store a collection of Note objects. The NoteFolder class has a single property named Notes that is of type ObservableCollection<Note>. The constructor calls an asynchronous method that obtains a list of all the files using FileHelper.GetFilesAsync, and then sorts them by filename. (Keep in mind that the filenames are constructed from the date and time when the Note object was created, so these filenames will be sorted into the order in which the files were created.) Then a Note object is created for each filename, the LoadAsync method is called to load the file into the Note object, and it's added to the Notes collection:

```
namespace NoteTaker8
{
    class NoteFolder
    {
        public NoteFolder()
        {
            this.Notes = new ObservableCollection<Note>();
            GetFilesAsync();
        }

        public ObservableCollection<Note> Notes { private set; get; }

        async void GetFilesAsync()
        {
            // Sort the filenames.
            IEnumerable<string> filenames =
                from filename in await FileHelper.GetFilesAsync()
                where filename.EndsWith(".note")
                orderby (filename)
                select filename;

            // Store them in the Notes collection.
```

```
            foreach (string filename in filenames)
            {
                Note note = new Note(filename);
                await note.LoadAsync();
                this.Notes.Add(note);
            }
        }
    }
}
```

This class does not implement `INotifyPropertyChanged` because it doesn't have to. The constructor instantiates an `ObservableCollection` and sets it to the `Notes` property and the `Notes` property never changes even though `Note` objects are later added to the collection.

To confirm to yourself that the objects are really being added to the ListView collection asynchronously, try inserting the following statement in the `foreach` loop:

```
await Task.Delay(1000);
```

That inserts a delay of 1 second but does not block the user-interface thread. It's rather fun to see the `ListView` updated with one item per second, and it's good for testing, but remember to remove such code before shipping the software!

The **NoteTaker8** program creates a single instance of the `NoteFolder` class and uses that instance throughout the program. Because it is convenient for this `NoteFolder` object to be accessible through the application, it is made available as a property of the `App` class:

```
namespace NoteTaker8
{
    public class App
    {
        static NoteFolder noteFolder = new NoteFolder();

        internal static NoteFolder NoteFolder
        {
            get { return noteFolder; }
        }

        public static Page GetMainPage()
        {
            return new NavigationPage(new HomePage());
        }
    }
}
```

Because there is only one instance of `NoteFolder` used throughout the application, it would make sense for `NoteFolder` to implement a singleton pattern, perhaps with a static property named `Instance` that insures that the `NoteFolder` is only instantiated once. That would be a sensible enhancement.

It's also possible to use `DependencyService` for this purpose because `DependencyService`

always returns a single cached instance of the implementation of an interface. Include the class that implements the interface you've defined in the PCL project rather than in the platform projects.

Notice that the `NoteFolder` property has an accessibility of `internal`. Because The `GetMainPage` method must be accessible from the three application projects, both `GetMainPage` and `App` must be public. However, there is no reason for the `NoteFolder` property to be accessible from the individual application projects, so the `internal` modifier limits its visibility to the PCL.

Here's the new `HomePage` class, which includes a `ListView` to display the existing notes:

```
namespace NoteTaker8
{
    class HomePage : ContentPage
    {
        public HomePage()
        {
            this.Title = "Note Taker 8";

            // Create and initialize ListView.
            ListView listView = new ListView
            {
                ItemsSource = App.NoteFolder.Notes,
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Create and initialize Button
            Button button = new Button
            {
                Text = "Add New Note",
                HorizontalOptions = LayoutOptions.Center
            };

            button.Clicked += (sender, args) =>
                {
                    // Create unique filename.
                    DateTime datetime = DateTime.UtcNow;
                    string filename = datetime.ToString("yyyyMMddHHmmssfff") + ".note";

                    // Navigate to new NotePage.
                    Note note = new Note(filename);
                    this.Navigation.PushAsync(new NotePage(note));
                };

            // Assemble page.
            this.Content = new StackLayout
            {
                Children =
                {
                    listView,
                    button
                }
            };
        }
```

```
    }
}
```

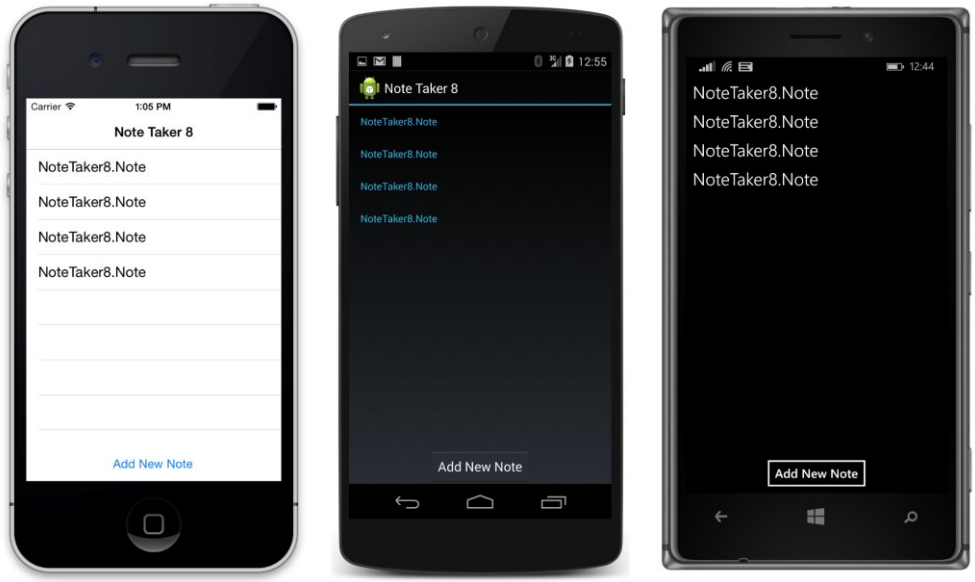
Notice that the `ItemsSource` property of the `ListView` is set to `App.NoteFolder.Notes`. The `ListView` is scrollable, so if it shares a `StackLayout` with some other view (such as `Button` on this page), it must have a `VerticalOptions` setting of `FillAndExpand`.

When the `Button` is clicked, `HomePage` constructs a unique filename, instantiates a `Note` object, and navigates to `NotePage`, which (as you've seen) sets its `BindingContext` to that `Note`. What you haven't seen yet is the new `OnDisappearing` override of `NotePage`:

```
namespace NoteTaker8
{
    class NotePage : ContentPage
    {

        …

        protected override void OnDisappearing()
        {
            // Only save it if there's some text somewhere.
            if (!String.IsNullOrWhiteSpace(note.Title) ||
                !String.IsNullOrWhiteSpace(note.Text))
            {
                note.SaveAsync();
                App.NoteFolder.Notes.Add(note);
            }
            base.OnDisappearing();
        }
    }
}
```

The new `OnDisappearing` saves the new `Note` object and adds it to the `App.NoteFolder.Notes` collection. As the `HomePage` comes back into view, the `ListView` shows the new `Note` object.

However, as you experiment with adding new `Note` objects, you'll find that the `ListView` display isn't all that illuminating:



This is probably not the last time you will see a list of fully-qualified class names in a `ListView`! But instead of feeling disheartened by such a display, you should be elated. This display clearly reveals that the `ListView` contains several objects of type `Note`. All you need do now is display them in some sensible manner.

Formatting items in a `ListView` is an art in itself, one that will be discussed in detail in a chapter not included in this Preview Edition. Right now, **NoteTaker8** will take an exceedingly simple approach. In the absence of any other formatting information, `ListView` uses the `ToString` method of the items to display them. These items are instances of the `Note` class, which does not have a `ToString` method, in which case the default `ToString` method simply displays the fully-qualified class name.

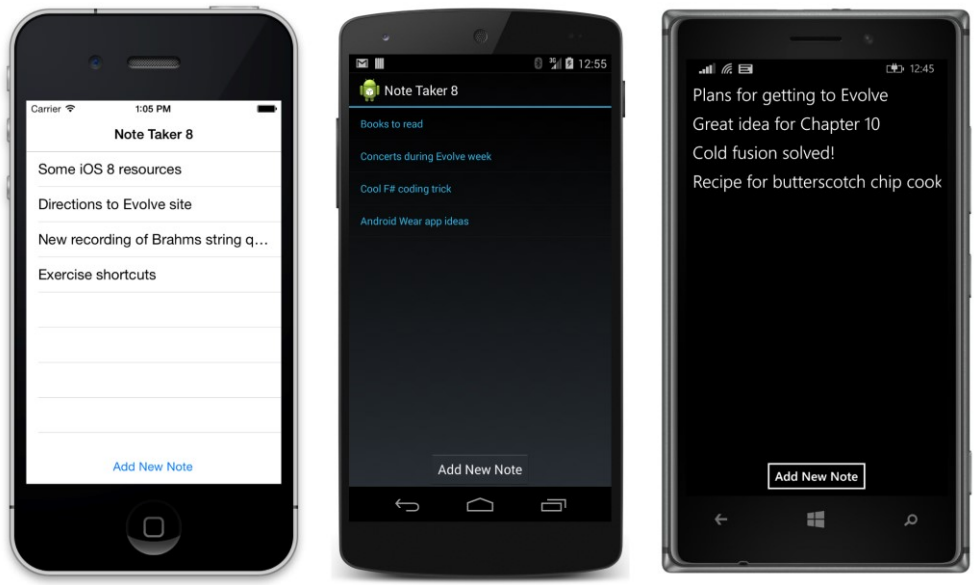
Let's add a `ToString` method to `Note`:

```
namespace NoteTaker8
{
    class Note : INotifyPropertyChanged
    {

        …

        public override string ToString()
        {
            if (!String.IsNullOrWhiteSpace(this.Title))
                return this.Title;

            int truncationLength = 30;

            if (this.Text.Length <= truncationLength)
                return this.Text;
```

```
            string truncated =
                this.Text.Substring(0, truncationLength);

            int index = truncated.LastIndexOf(' ');

            if (index != -1)
                truncated = truncated.Substring(0, index);

            return truncated;
        }
    }
}
```

Most of this method is dedicated to the cases where the user doesn't type in a title. The program really shouldn't allow a note without a title, but the philosophy these days is for programs to accommodate the whims of the user rather than the other way around. The `ToString` method attempts to use the first 30 characters of the note text but preferably truncated at a space.

Now the titles show up:



Version 9. Editing and deleting

Two essential features are missing from **NoteTaker8** and must be implemented in **NoteTaker9**:editing and deleting notes. This is not going to involve restructuring or refactoring. Much of the architecture is already in place.

You might have noticed that the `ListView` allows you to select items by tapping them. The item is

then displayed with platform-specific highlighting of some sort. In **NoteTaker9**, selecting an item signals a desire of the user to edit or delete a note.

When an item is selected, the `ListView` fires an `ItemSelected` event. The event arguments include a property named `SelectedItem` indicating the selected item. (The `ListView` itself also has a property named `SelectedItem`.) This `SelectedItem` property could be `null` if no item is currently selected. For that reason, `ItemSelected` handlers often check for a `null` value before proceeding.

Moreover, if you're using this selection to navigate to another page, you probably don't want the item still selected when the user returns to the page displaying the `ListView`. Because the `ListView` fires the `ItemSelected` event only when the `SelectedItem` property changes, nothing will happen when you tap an item that's already selected! For that reason, `ItemSelected` handlers used with page navigation usually set the `SelectedItem` property of the `ListView` to `null` before navigating to the other page. This process causes another `ItemSelected` event to fire but the `SelectedItem` property is now `null`. Checking for `null` values of `SelectedItem` allows the handler to avoid a problem with this recursive call.

Here's the definition of the `ListView` and its `ItemSelected` handler in **NoteTaker9**:

```
namespace NoteTaker9
{
    class HomePage : ContentPage
    {
        public HomePage()
        {

            …

            // Create and initialize ListView.
            ListView listView = new ListView
            {
                ItemsSource = App.NoteFolder.Notes,
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Handle item selection for editing and deleting.
            listView.ItemSelected += (sender, args) =>
                {
                    if (args.SelectedItem != null)
                    {
                        // Deselect the item.
                        listView.SelectedItem = null;

                        // Navigate to NotePage.
                        Note note = (Note)args.SelectedItem;
                        this.Navigation.PushAsync(new NotePage(note, true));
                    }
                };

            …
```

```
        }
    }
}
```

There are two `SelectedItem` properties here! The handler can obtain the selected `Note` object from the `SelectedItem` property of the event arguments, but it must deselect the selection by setting the `SelectedItem` property of the `ListView`. If you're using a named method for the handler, you can get access to the `ListView` object firing the event by casting the first argument to an object of type `ListView`.

The `NotePage` constructor referenced in the `ItemSelected` handler has a second argument that is set to `true`. The previous version of `NotePage` only had a single-argument constructor, so a second argument must be added to the constructor of the new `NotePage`, which is used to tailor the user interface and implement a little different logic when the page is being used to edit an existing note instead of to create a new one:

```
namespace NoteTaker9
{
    class NotePage : ContentPage
    {
        Note note;
        bool isNoteEdit;

        public NotePage(Note note, bool isNoteEdit = false)
        {
            this.note = note;
            this.isNoteEdit = isNoteEdit;
            Title = isNoteEdit ? "Edit Note" : "New Note";

            // Create Entry and Editor views.
            Entry entry = new Entry
            {
                Placeholder = "Title (optional)"
            };

            Editor editor = new Editor
            {
                Keyboard = Keyboard.Create(KeyboardFlags.All),
                BackgroundColor = Device.OnPlatform(Color.Default,
                                                    Color.Default,
                                                    Color.White),
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Set data bindings.
            this.BindingContext = note;
            entry.SetBinding(Entry.TextProperty, "Title");
            editor.SetBinding(Editor.TextProperty, "Text");

            // Assemble page.
            StackLayout stackLayout = new StackLayout
            {
```

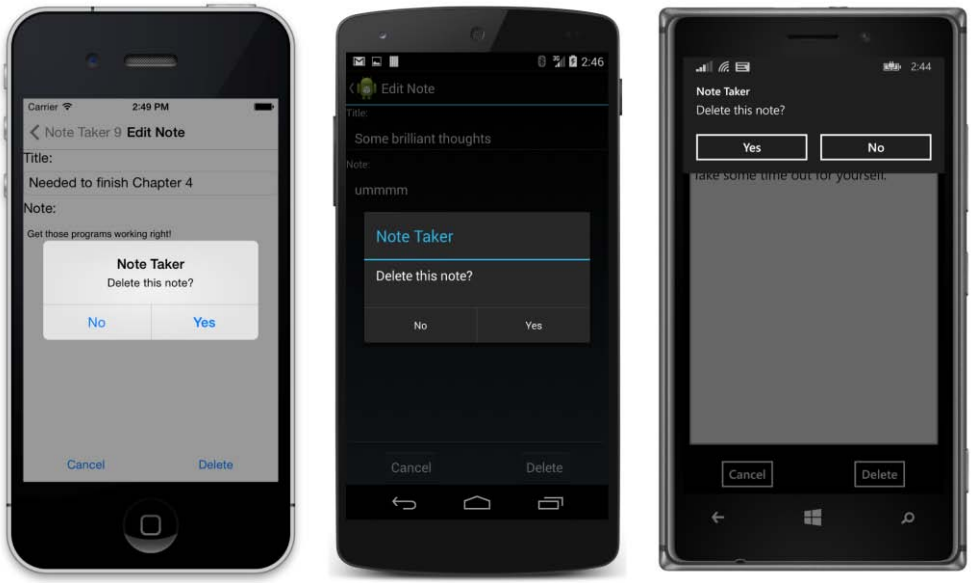
```
        Children =
        {
            new Label { Text = "Title:" },
            entry,
            new Label { Text = "Note:" },
            editor,
        }
    };

    if (isNoteEdit)
    {
        // Cancel button.
        Button cancelButton = new Button
        {
            Text = "Cancel",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };

        cancelButton.Clicked += async (sender, args) =>
            {
                bool confirm = await this.DisplayAlert("Note Taker",
                                                       "Cancel note edit?",
                                                       "Yes", "No");
                if (confirm)
                {
                    // Reload note.
                    await note.LoadAsync();

                    // Return to home page.
                    await this.Navigation.PopAsync();
                }
            };

        // Delete button
        Button deleteButton = new Button
        {
            Text = "Delete",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };

        deleteButton.Clicked += async (sender, args) =>
            {
                bool confirm = await this.DisplayAlert("Note Taker",
                                                       "Delete this note?",
                                                       "Yes", "No");
                if (confirm)
                {
                    // Delete Note file and remove from collection.
                    await note.DeleteAsync();
                    App.NoteFolder.Notes.Remove(note);

                    // Return to home page.
                    await this.Navigation.PopAsync();
                }
```

```
                };

            StackLayout horzStack = new StackLayout
            {
                Orientation = StackOrientation.Horizontal,
                Children =
                {
                    cancelButton,
                    deleteButton
                }
            };

            stackLayout.Children.Add(horzStack);
        }

        this.Content = stackLayout;
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();
    }

    protected override void OnDisappearing()
    {
        // Only save it if there's some text somewhere.
        if (!String.IsNullOrWhiteSpace(note.Title) ||
            !String.IsNullOrWhiteSpace(note.Text))
        {
            note.SaveAsync();

            if (!isNoteEdit)
            {
                App.NoteFolder.Notes.Add(note);
            }
        }

        base.OnDisappearing();
    }
  }
}
```

The **Cancel** and **Delete** buttons are created only if the `isNoteEdit` argument to the constructor is `true`. The **Cancel** button is used when you've navigated to `NotePage` to edit a note, but after typing in a little, you decide to abandon those edits. There is no **Ok** button to confirm your edits. That's just a matter of navigating back to the home page.

It's polite to obtain a confirmation from the user when deleting something or cancelling an operation. For confirmation purposes, the Xamarin.Forms `Page` class defines a `DisplayAlert` method, which displays a native platform alert box. Three different version of `DisplayAlert` exist. The one used for the **Cancel** and **Delete** options has two arguments to define the text of two buttons and returns `true` if the first button is tapped. `DisplayAlert` is asynchronous—which means that the

program code isn't blocked waiting for the alert to be dismissed—so it's commonly used with `await` to obtain the return value. Here's what it looks like on the three platforms:



If the user presses **Yes**, the `DisplayAlert` method returns `true`. The `Button` handler then carries out the operation and calls `PopAsync` on the page's `Navigation` property to return to the home page.

Deleting a note is equivalent to deleting the file associated with the note and removing it from the `Notes` collection in `NoteFolder`. For convenience, the `Note` class has a `DeleteAsync` method for this purpose. (Alternatively, `NotePage` could obtain the filename from the public `Filename` property of `Note` and delete the file itself.)

```
namespace NoteTaker9
{
    class Note : INotifyPropertyChanged
    {
        …
        public async Task DeleteAsync()
        {
            await FileHelper.DeleteFileAsync(this.Filename);
        }
        …
    }
}
```

Version 10. Adding a template cell

If you experiment with the editing facility of **NoteTaker9**, you might discover a bug. If you edit a title—or if a note doesn't have a title but you edit part of the text used for the title—you won't see the updated version of the note in the `ListView`. Terminate the program and start it up again, and your edits will appear.

The problem involves the simple approach taken to displaying the `Note` in the `ListView` using the `ToString` method. That's what `ListView` uses in the simple case, but it's not adequate. `ListView` calls the `ToString` method of every item when first displaying those items but is never informed when `ToString` might return a different value.

Fixing this problem requires a little deeper use of `ListView`.

To display the items in the `ItemsSource` collection, `ListView` uses an object called a *cell*. This cell is a view (or a tree of views) that serves as a template. The `ListView` uses this cell to generate the same view (or a tree of views) for each item in the collection. Normally, that would result in all the items being displayed in exactly the same way. However, this cell can contain data bindings that reference properties in the `ListView` items—the `Note` objects in this case.

You can make your own cells by using `ViewCell`, but several preexisting cells are available. The simplest is `TextCell`, which simply consists of a `Label`. The `ListView` uses a `TextCell` if no cell is specified, but the advantage of explicitly specifying a `TextCell` is that you can bind the `Text` property to a property in `Note`.

`Note` needs a new property to identify the object in the `ListView`. Let's call this property `Identifier`. The `Identifier` property will return the same string as the `ToString` method in the previous version of `Note`, but the big difference is that `Identifier` property also fires the `PropertyChanged` event.

If the `Title` property changes, the `Identifier` property must also change, and if the `Title` property is empty and the `Text` property changes, `Identifier` will change as well.

Here's the pertinent new logic in `Note`:

```
namespace NoteTaker10
{
    class Note : INotifyPropertyChanged
    {
        string title, text, identifier;

        …

        public string Title
        {
            set
            {
                if (SetProperty(ref title, value))
```

```
                {
                    this.Identifier = MakeIdentifier();
                }
            }
            get { return title; }
        }

        public string Text
        {
            set
            {
                if (SetProperty(ref text, value) &&
                    String.IsNullOrWhiteSpace(this.Title))
                {
                    this.Identifier = MakeIdentifier();
                }
            }
            get { return text; }
        }

        public string Identifier
        {
            private set { SetProperty(ref identifier, value); }
            get { return identifier; }
        }

        string MakeIdentifier()
        {
            if (!String.IsNullOrWhiteSpace(this.Title))
                return this.Title;

            int truncationLength = 30;

            if (this.Text == null ||
                this.Text.Length <= truncationLength)
            {
                return this.Text;
            }

            string truncated = this.Text.Substring(0, truncationLength);

            int index = truncated.LastIndexOf(' ');

            if (index != -1)
                truncated = truncated.Substring(0, index);

            return truncated;
        }

        …

    }
}
```

Keep in mind that the `SetProperty` method returns true if the property is changing, so the set accessors can use that return value to perform additional logic. The `Title` and `Text` properties use this return value to call `MakeIdentifier`. The logic is pretty much the same as the original `ToString` method. The `ToString` override has been removed. Although a class with a `ToString` method is often preferred to a class without one, when using `ListView` it's usually best to confirm that the template is working.

The other change involves the `ListView` in the new `HomePage`:

```
// Create and initialize ListView.
ListView listView = new ListView
{
    ItemsSource = App.NoteFolder.Notes,
    ItemTemplate = new DataTemplate(typeof(TextCell)),
    VerticalOptions = LayoutOptions.FillAndExpand
};

listView.ItemTemplate.SetBinding(TextCell.TextProperty, "Identifier");
```

`ListView` inherits both the `ItemsSource` and `ItemTemplate` properties from its parent class `ItemsView<T>`. This `ItemTemplate` property is of type `DataTemplate`, and the `DataTemplate` constructor requires a `Type` object of a class that derives from `Cell`. The `DataTemplate` uses this `Type` object to generate visual objects to display the `ListView` items.

For this `ListView`, the argument to the `DataTemplate` constructor is `typeof(TextCell)`. `TextCell` is basically a `Cell` wrapper around a `Label`, so for each item in the `ListView`, the `DataTemplate` object uses the `TextCell` to generate a `Label` to display that item. This is the simplest type of template; the final version of this book will show how to create more extensive templates.

Here's the important part: Although `DataTemplate` does not derive from `BindableObject`, you can still set data bindings on it. `DataTemplate` defines its own `SetBinding` method. The data binding target must be a property of the cell, in this case the `Text` property defined by `TextCell`. The data-binding source is a property of the items in the `ListView`, in this case a property of the `Note` class, specifically, `Identifier`.

When the `Label` objects are generated to render each item in the `ListView`, this binding is transferred from the `DataTemplate` to the `Text` property of the `Label` object. The `BindingContext` of each `Label` is set to the `Note` object associated with that item.

The visuals of **NoteTaker10** are the same as **NoteTaker9**, but when you change the title of the note—or when you change the text if the title is empty—the text displayed by the `ListView` changes as well.

Bug eliminated.

Version 11. Constructing a toolbar

`HomePage` has an **Add New Note** button, and `NotePage` has two buttons for **Cancel** and **Delete**. These shouldn't really be buttons. They should be toolbar items. You can add toolbar items to a `ContentPage` in Windows Phone, but iOS and Android restrict toolbar items to a `NavigationPage` or a page navigated to from a `NavigationPage`.

The `ToolbarItem` class does not derive from `View` like `Button` and `ListView`. It's really just a class for defining the characteristics of a toolbar item using the following properties:

- `Name` — the text that might appear
- `Icon` — the filename of a bitmap (most commonly PNG) stored as a resource
- `Order` — a member of the `ToolbarItemOrder` enumeration: `Default`, `Primary`, or `Secondary`
- `Priority` — an integer

The `Order` property governs whether the `ToolbarItem` appears as an image (`Primary`) or text (`Secondary`). The Windows Phone is limited to four `Primary` items, and both the iPhone and Android start getting crowded with more than that, so that's a reasonable limitation. Additional `Secondary` items are text only. On the iPhone they appear underneath the `Primary` items; on Android and Windows Phone they aren't seen on the screen until the user taps a vertical or horizontal ellipsis.

The `Icon` property is crucial for `Primary` items, of course, and the `Name` property is crucial for `Secondary` items, but Windows Phone also uses the `Name` to display a short text hint underneath the icons for `Primary` items.

When the `ToolbarItem` is tapped, it fires an `Activated` event rather than a `Clicked` event like the `Button` but the concept is the same. (`ToolbarItem` also has `Command` and `CommandParameter` properties like the `Button`; these are for data binding purposes and will be demonstrated in a later chapter not in this Preview Edition.)

The `ToolbarItem` constructor lets you specify all these properties (with the exception of `Command` and `CommandParameter`) along with a handler for the `Activated` event.

An instance of `ToolbarItem` cannot be added to a `StackLayout` like a view. It must be added to the `ToolbarItems` property of `Page`. This `ToolbarItems` property is of type `IList<ToolbarItem>`. Once you add a `ToolbarItem` to this collection, the `ToolbarItem` properties cannot be changed. The property settings are instead used internally to construct platform-specific objects.

Certainly the most difficult aspect of using `ToolbarItem` is assembling the bitmap images for the `Icon` property. Each platform has different requirements for the color composition and size of these icons, and each platform has somewhat different conventions for the imagery. The standard icon for

Share, for example, is different on Android and Windows Phone.

For these reasons, it makes sense for each of the platform projects to have its own collection of toolbar icons rather than to use a shared set of icons in the common PCL project. The PCL project then references these platform images.

Let's begin with the two platforms that provide collections of icons suitable for `ToolbarItem`.

Icons for Android

The Android website has a downloadable collection of toolbar icons at this URL:

http://developer.android.com/design/downloads

Download the ZIP file identified as Action Bar Icon Pack.

The unzipped contents are organized into two main directories: *Core_Icons* (23 images), and *Action Bar Icons* (144 images). These are all PNG files, and the *Action Bar Icons* come in four different sizes indicated by the directory name:

drawable-mdpi (medium dots per inch) — 32 pixels square

drawable-hdpi (high dots per inch) — 48 pixels square

drawable-xhdpi (extra high DPI) — 64 pixels square

drawable-xxhdpi (extra extra high DPI) — 96 pixels square

These directory names are special. They are the same names as the folders you should use in your Android project for storing the icon bitmaps you need. If you supply different sizes of icons in these directories, a size appropriate for the particular device will be selected at run time.

The *Core_Icons* folder also arranges its icons into four directories with the same four sizes, but these directories are named *mdpi*, *hdpi*, *xdpi*, and *unscaled*.

The *Action Bar Icons* folder has an additional directory organization using the names *holo_dark* and *holo_light*. Generally for Android, you'll want to use PNG files that have a white foreground on a transparent background; this is the color composition of the files in the *holo_dark* folder, meaning they are designed for a dark background. The *holo_light* icons have black foregrounds on a transparent background; they are much easier to see in **Finder** and **Windows Explorer**, but for most purposes you should use the *holo_dark* icons.

The *Core_Icons* folder only contains icons with white foregrounds on a transparent background.

If you are reproducing **NoteTaker11** yourself, use Visual Studio or Xamarin Studio to first create four subfolders in the *Resources* folder of the Android project: *drawable-mdpi*, *drawable-hdpi*, *drawable-xhdpi*, and *drawable-xxhdpi*. (In both Visual Studio and Xamarin Studio, the context menu item is **Add** and **New Folder**.) Then add existing bitmap files to those folders. In Visual Studio the context menu item is **Add** and **Existing Item**. In Xamarin Studio it's **Add** and **Add Files**. You'll want to

copy from the following directories in *holo_dark*, four sizes of icons with the following names to the corresponding project folders:

- from *01_core_new*, four sizes of ic_action_new.png
- from *01_core_cancel*, four sizes of ic_action_cancel.png
- from *01_core_discard*, four sizes of ic_action_discard.png

Yes, it can be tedious.

Check the properties of these PNG files. They must have a **Build Action** of **AndroidResource**.

Icons for Windows Phone

If you have a version of Visual Studio installed for Windows Phone 8, you can find a collection of PNG files suitable for `ToolbarItem` in the following directory on your hard drive:

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v8.0\Icons

There are two subdirectories, *Dark* and *Light*, each containing the same 37 images. As with Android, the icons in the *Dark* directory have white foregrounds on transparent backgrounds, and the icons in the *Light* directory have black foregrounds on transparent backgrounds. You should use the ones in the *Dark* directory.

The images are a uniform 76 pixels square but have been designed to appear inside a circle. Indeed, one of the files is named basecircle.png that can serve as a guide if you'd like to design your own, so there are really only 36 usable icons in the collection and a couple of them are the same.

Generally in a Windows Phone project, image files such as these are stored in a separate folder, usually *Assets* (which already exists in the Windows Phone project) or *Images*. You can make an *Images* folder and then add the following bitmaps to that folder:

- add.png
- cancel.png
- delete.png

Check the properties of these files, and make sure the **Build Action** is **Content**.

Icons for iOS devices

This is the most problematic platform for `ToolbarItem`. If you're programming directly for the native iOS API, a bunch of constants let you select an image for `UIBarButtonItem`, which is the underlying iOS implementation of `ToolbarItem`. But for the Xamarin.Forms `ToolbarItem` you'll need to obtain icons from another source—perhaps licensing a collection—or make your own.

For best results, you should supply two image files for each toolbar item in a folder names

Resources. An image with a filename such as image.png should be 20 pixels square, while the same image should also be supplied in a 40 pixel square dimension with the name image@2x.png. At run time, iOS selects which to use based on the resolution of the device. Retina devices get the larger image. If you supply image files of other dimensions, some scaling might be performed. Both bitmaps should have a black foreground on a transparent background. The operating system will color the foreground appropriately.

Here's a collection of free unrestricted-use icons used for the program in Chapter 1:

http://www.smashingmagazine.com/2010/07/14/gcons-free-all-purpose-icons-for-designers-and-developers-100-icons-psd/

However, they are uniformly 32-pixels square, and some basic ones are missing.

The iOS project in **NoteTaker11** uses Android icons from the *holo_light* directories *mdpi* and *hdpi*. These were added to the **Resources** directory of the iOS project and renamed, with the assumption that iOS would perform the size conversions at run time:

- new.png and new@2x.png
- cancel.png and cancel@2x.png
- discard.png and discard@2x.png

For toolbar icons in an iOS project, the **Build Action** must be **BundleResource**.

ToolbarItem code

Once the icons have been added to the three projects, writing the code for `ToolbarItem` is definitely the easier part of the process!

The code required in `HomePage` for the `ToolbarItem` is quite similar to the previous `Button` logic. Like the `Button`, the `ToolbarItem` must be instantiated and initialized. The handler for the `Activated` event for the `ToolbarItem` is identical to the handler for the `Clicked` event of the `Button`. The only big difference is that the `ToolbarItem` must be added to the `Toolbars` collection of the page. Here's the entire new version of `HomePage`:

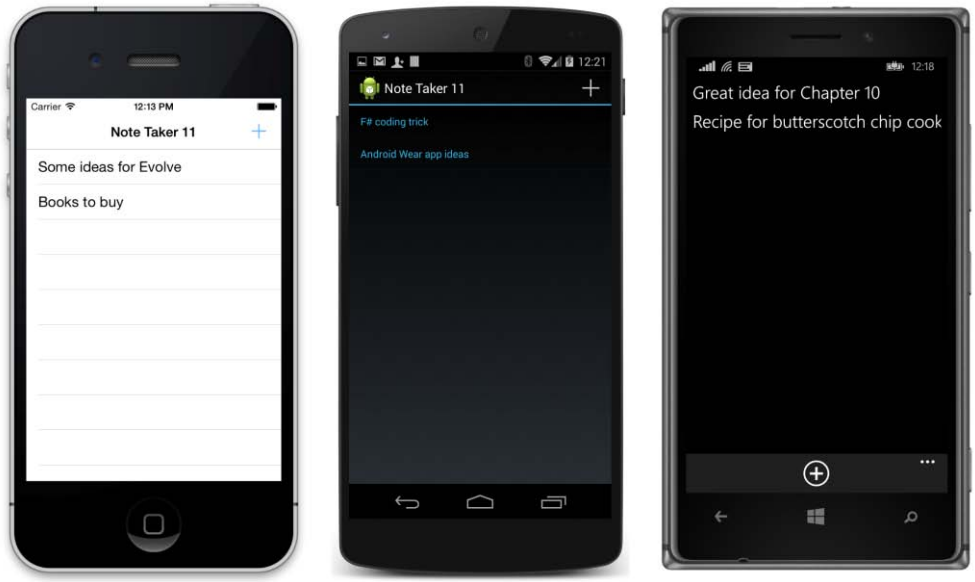
```
namespace NoteTaker11
{
    class HomePage : ContentPage
    {
        public HomePage()
        {
            this.Title = "Note Taker 11";

            // Create and initialize ListView.
            ListView listView = new ListView
            {
                ItemsSource = App.NoteFolder.Notes,
                ItemTemplate = new DataTemplate(typeof(TextCell)),
```

```
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        listView.ItemTemplate.SetBinding(TextCell.TextProperty, "Identifier");

        // Handle item selection for editing and deleting.
        listView.ItemSelected += (sender, args) =>
            {
                if (args.SelectedItem != null)
                {
                    // Deselect the item.
                    listView.SelectedItem = null;

                    // Navigate to NotePage.
                    Note note = (Note)args.SelectedItem;
                    this.Navigation.PushAsync(new NotePage(note, true));
                }
            };

        // Create and initialize ToolbarItem.
        ToolbarItem addNewItem = new ToolbarItem
        {
            Name = "Add Note",
            Icon = Device.OnPlatform("new.png",
                                     "ic_action_new.png",
                                     "Images/add.png"),
            Order = ToolbarItemOrder.Primary
        };

        addNewItem.Activated += (sender, args) =>
            {
                // Create unique filename.
                DateTime datetime = DateTime.UtcNow;
                string filename = datetime.ToString("yyyyMMddHHmmssfff") + ".note";

                // Navigate to new NotePage.
                Note note = new Note(filename);
                this.Navigation.PushAsync(new NotePage(note));
            };

        this.ToolbarItems.Add(addNewItem);

        // Assemble page.
        this.Content = listView;
    }
  }
}
```

Notice the `Device.OnPlatform` method used to obtain the correct image from the three platform projects. Including the folder with the filename is required for Windows Phone but not for iOS or Android because those icons must be stored in specific folders.

You can see the **Add** icon in the upper right corner of the iOS and Android screens and at the

bottom of the Windows Phone screen:



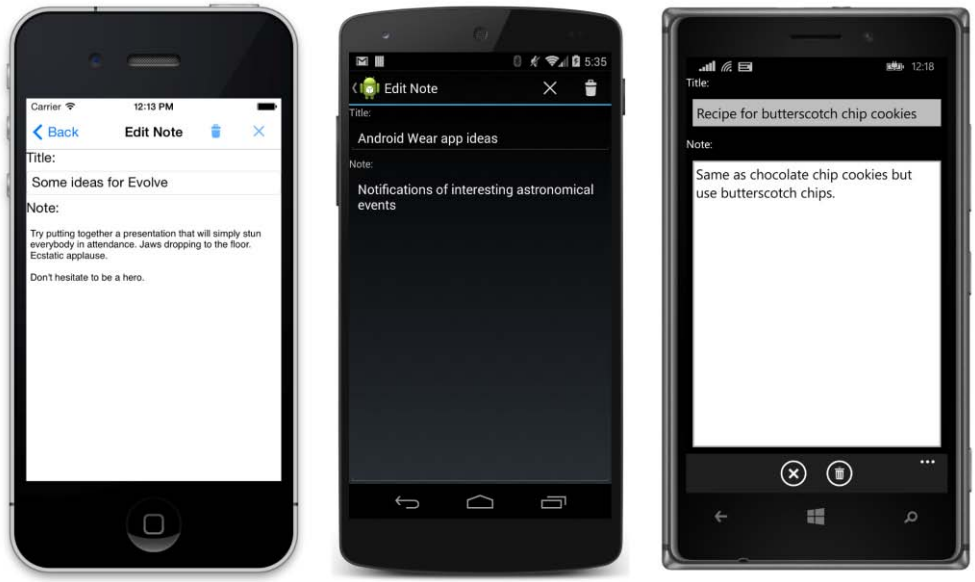
If you press the ellipsis at the lower right corner of the Windows Phone screen, the Windows Phone application bar slides up a little to display text under the icon, in this case "Add Note", which is the string set to the `Name` property of the `ToolbarItem` object.

The `NotePage` has two `ToolbarItem` objects. The constructor adds these to the `ToolbarItems` collection of the page only if the second argument to the constructor is `true`:

```
namespace NoteTaker11
{
    class NotePage : ContentPage
    {
        Note note;
        bool isNoteEdit;

        public NotePage(Note note, bool isNoteEdit = false)
        {

            …

            if (isNoteEdit)
            {
                // Cancel toolbar item.
                ToolbarItem cancelItem = new ToolbarItem
                {
                    Name = "Cancel",
                    Icon = Device.OnPlatform("cancel.png",
                                             "ic_action_cancel.png",
                                             "Images/cancel.png"),
                    Order = ToolbarItemOrder.Primary
```

```
            };

            cancelItem.Activated += async (sender, args) =>
                {
                    bool confirm = await this.DisplayAlert("Note Taker",
                                                           "Cancel note edit?",
                                                           "Yes", "No");
                    if (confirm)
                    {
                        // Reload note.
                        await note.LoadAsync();

                        // Return to home page.
                        await this.Navigation.PopAsync();
                    }
                };

            this.ToolbarItems.Add(cancelItem);

            // Delete toolbar item.
            ToolbarItem deleteItem = new ToolbarItem
            {
                Name = "Delete",
                Icon = Device.OnPlatform("discard.png",
                                         "ic_action_discard.png",
                                         "Images/delete.png"),
                Order = ToolbarItemOrder.Primary
            };

            deleteItem.Activated += async (sender, args) =>
                {
                    bool confirm = await this.DisplayAlert("Note Taker",
                                                           "Delete this note?",
                                                           "Yes", "No");
                    if (confirm)
                    {
                        // Delete Note file and remove from collection.
                        await note.DeleteAsync();
                        App.NoteFolder.Notes.Remove(note);

                        // Return to home page.
                        await this.Navigation.PopAsync();
                    }
                };

            this.ToolbarItems.Add(deleteItem);
        }

        this.Content = stackLayout;
    }

    …
}
}
```

The `NotePage` displays these two items and implements them pretty much like the previous `Button` views:



Version 12. Application lifecycle events

Suppose someone is using the **NoteTaker** program and decides to enter a new note or edit an existing note. The user enters some text in the `Entry` and `Editor` views, and then is interrupted with a distraction—perhaps a phone call. Later on, the user shuts off the phone.

What should happen the next time the **NoteTaker** program is launched? Should that new note in progress—or the edits to the existing note—have been discarded? That doesn't seem reasonable, yet if you consider that **NoteTaker** saves notes only during the `OnDisappearing` override of the `NotePage` class, that's exactly what will happen.

What can be done about this problem? One approach is to attach handlers for the `TextChanged` events of the `Entry` and `Editor` views and save the file every time the text changes. But that's a lot of file activity, and you'd need to have some special provision if the user decides to cancel the edits.

This problem is not something that programmers worried about in the old days. Programs on desktop computers have specific commands to open and save files, and if you as a user try to close a program without saving the document first, the program asks you if you want to save your work. But if you happen to switch off your computer without saving first, that's your problem!

Expectations are very different with mobile devices. Users expect applications to remember exactly what they were doing the last time they interacted with the program.

For this reason, all three platforms supported by Xamarin.Forms include programming interfaces that allow developers to save interim data and program state at crucial times during the lifetime of the application. These APIs are described generally under the category of *application lifecycle* or (for the Android operating system) *activity lifecycle*.

The Xamarin documentation for iOS discusses application lifecycle issues in the context of backgrounding:

http://developer.xamarin.com/guides/android/application_fundamentals/activity_lifecycle/

Applications are said to be *activated* when they launch and *terminated* when they end. In between those two events, an application might *enter the background*. For example, pressing the **Home** button below the iPhone screen causes the application to enter the background during the time the user interacts with the home screen. Returning to the application causes the app to re-enter the foreground. However, the app might be terminated before that happens.

The Xamarin documentation for Android covers activity lifecycle here:

http://developer.xamarin.com/guides/android/application_fundamentals/activity_lifecycle/

An activity is said to be *active* or *running* when it first starts up but can later be *paused* if the user presses the **Home** button (or performs other actions that move the application from the screen). From that point the activity can be *resumed* or *stopped*. From the stopped state, the activity must either be restarted or destroyed.

Microsoft documents the application lifecycle for Windows Phone 8 here:

http://msdn.microsoft.com/en-us/library/windows/apps/ff817008(v=vs.105).aspx

After being launched, an application is normally running, but it can be *deactivated* in various ways, including the user pressing the **Home** button. The application can later be re-activated from the deactivated state, or it might be terminated.

All three platforms include the concept of a program being paused, or suspended, or backgrounded (whatever it's called) during certain types of interaction that cause the program to be removed from the screen. From that state, the application might be terminated, or it might be resumed (that is, moved to the foreground), and continue interacting with the user from the point where it was paused.

It is possible for a developer to get very deep into the various protocols and notifications that support application lifecycle handling. The approach taken here really just covers the simplest of cases.

The **NoteTaker** application only has a problem with the following sequence of events:

1. The user navigates to the `NotePage` for entering a new note or editing an existing note.
2. Some text is typed into the `Entry` or `Editor`.
3. The program is paused or suspended in some way. Perhaps the user presses the hardware

Home button.

4. The program is terminated before it has a chance to be resumed.

The next time the program starts up, it should automatically open to the `NotePage` and the `Entry` and `Editor` contents should be restored to the same state as they were when the program was previously suspended.

How should **NoteTaker** handle this?

Let's assume **NoteTaker** has access to two system events named `Suspending` and `Resuming`. In its `OnAppearing` override, `NotePage` attaches handlers for these two events. The program is interested in these events only when `NotePage` is displayed. If the program is suspended, the handler for the `Suspending` event kicks in and saves the following information in a special file named "TransientData.save" (or something like that):

- The filename of the note. This is available from the `Note` object.
- This `isNoteEdit` flag originally passed to the `NotePage` constructor.
- The contents of the `Entry` view.
- The contents of the `Editor` view.

If the program is resumed, the code in the `Resuming` event handler runs. This code checks to see if a file named "TransientData.save" exists and, if so, deletes it. Because `NotePage` has resumed execution, that file is no longer needed.

The `OnDisappearing` override of `NotePage` detaches the `Suspending` and `Resuming` event handlers. The handlers are only needed in `NotePage`, and detaching them allows `NotePage` to be disposed of by the operating system during garbage collection.

The end result is this: If the app was suspended while `NotePage` was active, and if the app was never resumed, a file exists in the application's private storage area named "TransientData.save".

When **NoteTaker** starts up, `HomePage` needs to check for the existence of this file. If it exists, then `HomePage` should load the contents of the file and then delete the file so it won't be detected the next time the program starts up. `HomePage` should create a new `Note` object but initialize it with the saved contents of the `Entry` and `Editor`. It can then navigate directly to `NotePage`, which should then be in the same state as when it was so rudely interrupted.

This sounds like a strategy! But it's going to require another set of helpers in the `Platform-Helpers` namespace to get platform-specific `Resuming` and `Suspending` events. Here's the `ILifecycleHelper` interface in the **NoteTaker12** project:

```
namespace PlatformHelpers
{
    public interface ILifecycleHelper
    {
```

```
        event Action Suspending;

        event Action Resuming;
    }
}
```

Just two events with the simplest type of event handler: an `Action` with no arguments and no return value.

The iOS implementation

The constructor of the iOS implementation of this interface sets handlers for the iOS `ObserveDidEnterBackground` and `ObserveWillEnterForeground` notifications. When these notifications occur, the handlers fire `Suspending` and `Resuming` events, respectively:

```
[assembly: Dependency(typeof(PlatformHelpers.iOS.LifecycleHelper))]

namespace PlatformHelpers.iOS
{
    class LifecycleHelper : ILifecycleHelper
    {
        public event Action Suspending;

        public event Action Resuming;

        public LifecycleHelper()
        {
            UIApplication.Notifications.ObserveDidEnterBackground(
                (sender, args) =>
                {
                    if (Suspending != null)
                        Suspending();
                });

            UIApplication.Notifications.ObserveWillEnterForeground(
                (sender, args) =>
                {
                    if (Resuming != null)
                        Resuming();
                });
        }
    }
}
```

Don't forget the `Dependency` attribute!

The Android implementation

The Android implementation is a little messier. The Android `Activity` class defines virtual protected `OnPause` and `OnResume` methods, so you'll need to override these methods in the `MainActivity` class created as part of the standard Xamarin.Forms solution template. These overrides should then call

methods in the `ILifecycleHelper` implementation class, which can then fire the `Suspending` and `Resuming` events.

That sounds straightforward, but as soon as you start coding it, a problem arises: In order to use the `LifecycleHelper`, the `NotePage` in the PCL will use the `DependencyService` class to instantiate this Android implementation of the `LifecycleHelper` class at run time. Thus, the PCL project has access to that instance but the Android project itself does not! This means that the `OnPause` and `OnResume` overrides in `MainActivity` need to call *static* methods of the `LifecycleHelper` class, and then these static methods need to fire events on the instance of `LifecycleHelper` created by `DependencyService`.

For this reason, whenever the Android version of `LifecycleHelper` is instantiated, it saves that instance in a private static field. (The `DependencyService` only instantiates the `LifecycleHelper` class once and then caches the result.) The static methods called from `MainActivity` access that instance to fire the events:

```
//-------------------------------------------------------
// This Android LifecycleHelper class requies calls from
// OnPause and OnResume overrides in MainActivity to the
// static OnPause and OnResume in this class!
//-------------------------------------------------------

[assembly: Dependency(typeof(PlatformHelpers.Droid.LifecycleHelper))]

namespace PlatformHelpers.Droid
{
    class LifecycleHelper : ILifecycleHelper
    {
        public event Action Suspending;

        public event Action Resuming;

        static LifecycleHelper instance;

        public LifecycleHelper()
        {
            instance = this;
        }

        public static void OnPause()
        {
            if (instance != null && instance.Suspending != null)
                instance.Suspending();
        }

        public static void OnResume()
        {
            if (instance != null && instance.Resuming != null)
                instance.Resuming();
        }
    }
```

```
}
```

The file contains a comment that reminds us that including this file in the Android project is not sufficient. The `MainActivity` class must be modified to make calls into this class. Here is that `MainActivity` class in **NoteTaker12**:

```
namespace NoteTaker12.Droid
{
    [Activity(Label = "NoteTaker12", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : AndroidActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            Xamarin.Forms.Forms.Init(this, bundle);

            SetPage(App.GetMainPage());
        }

        protected override void OnPause()
        {
            PlatformHelpers.Droid.LifecycleHelper.OnPause();
            base.OnPause();
        }

        protected override void OnResume()
        {
            PlatformHelpers.Droid.LifecycleHelper.OnResume();
            base.OnResume();
        }
    }
}
```

The Windows Phone implementation

The Windows Phone version of `LifecycleHelper` is similar to the iOS version in that it uses the `LifecycleHelper` constructor to install handlers for some system events, which are then handled by firing the events defined in the interface:

```
[assembly: Dependency(typeof(PlatformHelpers.WinPhone.LifecycleHelper))]

namespace PlatformHelpers.WinPhone
{
    class LifecycleHelper : ILifecycleHelper
    {
        public event Action Suspending;

        public event Action Resuming;

        public LifecycleHelper()
        {
```

```
            PhoneApplicationService.Current.Launching +=
                (sender, args) =>
                {
                    if (Resuming != null)
                        Resuming();
                };

            PhoneApplicationService.Current.Activated +=
                (sender, args) =>
                {
                    if (Resuming != null)
                        Resuming();
                };

            PhoneApplicationService.Current.Deactivated +=
                (sender, args) =>
                {
                    if (Suspending != null)
                        Suspending();
                };

            PhoneApplicationService.Current.Closing +=
                (sender, args) =>
                {
                    if (Suspending != null)
                        Suspending();
                };
        }
    }
}
```

It seems as if only the `Activated` and `Deactivated` events need to be handled here, but that's not quite the case. When an application is started up, it gets either a `Launching` or `Activated` event but not both, and when a program is shut down, it gets either a `Deactivated` or `Closing` event but not both. The program should save interim data if either the `Deactivated` or `Closing` event occurs. The `Activated` event occurs following a previous `Deactivated` event, so the `Launching` event isn't strictly required, but it's included here for completeness. The `Closing` event is included here for completeness as well: The `Closing` event indicates that the user has pressed the hardware **Back** button to terminate an application, and in this program that can occur only when `HomePage` is active.

Saving and restoring

Both `HomePage` and `NotePage` need access to a filename used for storing the transient data. A convenient place to define this is the `App` class:

```
namespace NoteTaker12
{
    public class App
    {
        static NoteFolder noteFolder = new NoteFolder();
```

```
        internal static readonly string TransientFilename = "TransientData.save";

        internal static NoteFolder NoteFolder
        {
            get { return noteFolder; }
        }

        public static Page GetMainPage()
        {
            return new NavigationPage(new HomePage());
        }
    }
}
```

The constructor of `NotePage` is unchanged. However, the `OnAppearing` override needs to attach handlers for the `Suspending` and `Resuming` events, and the `OnDisappearing` override must detach those handlers. The `OnSuspending` handler uses a single string to assemble all the information it needs to restore the contents of the page. (The four items are separated by the ASCII '\x1F' character, which is a character that won't appear in the text itself and which is actually defined in the ASCII specification as a **Unit Separator**. Obviously, once program data gets much more complex than this, you'll want to use a more structured way to store it, such as XML or JSON.) The string is then saved with a call to `WriteTextAsync`. If the program is resumed, that data is not needed, so the `OnResuming` handler deletes the file:

```
namespace NoteTaker12
{
    class NotePage : ContentPage
    {
        Note note;
        bool isNoteEdit;
        ILifecycleHelper helper = DependencyService.Get<ILifecycleHelper>();

        …

        protected override void OnAppearing()
        {
            // Attach handlers for Suspending and Resuming event.
            helper.Suspending += OnSuspending;
            helper.Resuming += OnResuming;

            base.OnAppearing();
        }

        protected override void OnDisappearing()
        {
            // Only save it if there's some text somewhere.
            if (!String.IsNullOrWhiteSpace(note.Title) ||
                !String.IsNullOrWhiteSpace(note.Text))
            {
                note.SaveAsync();

                if (!isNoteEdit)
```

```
                {
                    App.NoteFolder.Notes.Add(note);
                }
            }

            // Detach handlers for Suspending and Resuming events.
            helper.Suspending -= OnSuspending;
            helper.Resuming -= OnResuming;

            base.OnDisappearing();
        }

        void OnSuspending()
        {
            // Save transient data and state.
            string str = note.Filename + "\x1F" +
                         isNoteEdit.ToString() + "\x1F" +
                         note.Title + "\x1F" +
                         note.Text;

            // Run entire WriteTextAsync in separate thread.
            Task task = Task.Run(() =>
                {
                    FileHelper.WriteTextAsync(App.TransientFilename, str);
                });

            // Wait for it to finish before finishing event handler.
            task.Wait();
        }

        async void OnResuming()
        {
            // Delete transient data and state.
            if (await FileHelper.ExistsAsync(App.TransientFilename))
            {
                await FileHelper.DeleteFileAsync(App.TransientFilename);
            }
        }
    }
}
```

But take a look at how the `OnSuspending` handler calls `FileHelper.WriteTextAsync`: It defines a lambda function with that call and passes it to a to a `Task.Run` creation method that returns a `Task` object, and then it calls `Wait` on that object. What's that all about?

This takes care of a problem that manifests itself in Windows Phone. When Windows Phone is suspending a program, the current `PhoneApplicationService` fires a `Deactivated` event. When the handler for that `Deactivated` event returns, the Windows Phone operating system assumes that the application has done everything it needs to do and will then suspend the app's activities. Sounds reasonable, right?

However, if the `Deactivated` handler is calling asynchronous methods—such as are required for

writing program data to a file—then the `Deactivated` handler returns control to the operating system at the first `await` operator or the first assignment of a `Completed` handler. The process of saving the file is still going on in the background, but these operations won't complete because the program is now suspended.

To get around that problem, the `OnSuspending` handler passes the call to `FileHelper.WriteTextAsync` to a `Task.Run` creation method. This causes the entire `WriteTextAsync` operation to run in a separate thread of execution. The `Wait` call effectively waits for that entire operation to complete. Because `WriteTextAsync` is not running in the user-interface thread, other code can run in the user-interface thread during this time, but the `OnSuspending` handler is blocked from concluding until `WriteTextAsync` has concluded..

`HomePage` must check for the existence of that special file and navigate to `NotePage` if it exists. Because asynchronous operations are involved and constructors can't have an `async` modifier, this job is relegated to a separate method named `CheckForSavedInfo`. This method reads the string from the file, breaks it down into the component pieces, creates a `Note` object, initializes the `Title` and `Text` properties with the saved text, and then navigates to `NotePage`:

```
namespace NoteTaker12
{
    class HomePage : ContentPage
    {
        public HomePage()
        {

            …

            CheckForSavedInfo();
        }

        async void CheckForSavedInfo()
        {
            if (await FileHelper.ExistsAsync(App.TransientFilename))
            {
                // Read the file.
                string str = await FileHelper.ReadTextAsync(App.TransientFilename);

                // Delete the file.
                await FileHelper.DeleteFileAsync(App.TransientFilename);

                // Break down the file contents.
                string[] contents = str.Split('\x1F');
                string filename = contents[0];
                bool isNoteEdit = Boolean.Parse(contents[1]);
                string entryText = contents[2];
                string editorText = contents[3];

                // Create the Note object and initialize it with saved data.
                Note note = new Note(filename);
                note.Title = entryText;
                note.Text = editorText;
```

```
                // Navigate to NotePage.
                NotePage notePage = new NotePage(note, isNoteEdit);
                await this.Navigation.PushAsync(notePage);
            }
        }
    }
}
```

`NotePage` should then be restored to the same state as when the user left it.

`DependencyService` is not the only way to deal with application lifecycle issues in a Xamarin.Forms application. Because the PCL does not need to make calls into the platform projects, the `MessageCenter` class can also provide a means for the platforms to signal application lifecycle activity to the shared project.

Some final advice: Dealing with application lifecycle issues is an important part of mobile development, and you'll have best results if you plan ahead. You should build application lifecycle handling into the application from the very beginning. Don't leave it to the end. If you're getting around to handling application lifecycle issues in the twelfth version of your program, you're doing it wrong. It's going to feel like a hack, and it might even appear like a hack.

Making it pretty

The source code for this chapter includes a final version of the program called simply **NoteTaker**. Screenshots from this version appear at the beginning of Chapter 3. This version was given an application icon, which is used in the start screens on the phon, and which you can also see in the upper-left corner of the Android page in those first three screenshots. In addition, the **NoteTaker** program appears in the start screens of the three platforms with the name "Note Taker" with two words rather than "NoteTaker".

These represent the first couple steps you should take when preparing an application for the market. There is much more to be done, of course, but that really comes under the category of iOS development, and Android development, and Windows Phone development rather than Xamarin.Forms development.

The application icon for **NoteTaker** was designed in Sketch 3 (a popular drawing program for the Mac) and exported to a bunch of different sizes.

If you are preparing an iOS application specifically for iPhone and not the iPad, it should have application icons of five different sizes added to the *Resources* folder. The filenames indicate the size: Icon-29.png, Icon-57.png, Icon-58.png, Icon-114.png, and Icon-120.png. These should have a **Build Action** of **BundleResource**.

In the Info.plist file, in the **iPhone Deployment Info** section in Visual Studio (or the **iOS Application Target** section in Xamarin Studio), you can specify an **Application Name** that is a little

different from the assembly name, in this case "Note Taker" with a space.

For an Android project, you can set a program icon by adding files named icon.png of various sizes to the following subfolders of *Resources*:

- *drawable-mdpi* — 48 pixels square
- *drawable-hdpi* — 72 pixels square
- *drawable-xhdpi* — 96 pixels square
- *drawable-xxhdpi* — 144 pixels square

Specify a **Build Action** of **AndroidResource**.

In Visual Studio, in the **Android Manifest** page of project properties, select @drawable/icon in the **Application Icon** section. In Xamarin Studio, in the **Options** page of the Android project, in the **Build** section and **Android Application** screen, use the **Application Icon** field to select the icon name.

You can specify a friendlier Android application name in the `Label` field of the `Activity` attribute on the `MainActivity` class.

The *Assets* folder of a Windows Phone application should have an ApplicationIcon.png file of 100 pixels square. This is the icon used in the application list. The **Build Action** is **Content**. The *Assets/Tile* folder should at least have icon files named FlipCycleTitleSmall.png (of 159 pixels square) and FlipCycleTitleMedium.png (of 336 pixels square) for programs that are pinned to the start screen.

In the **Application UI** section of the WMAppManifest.xml file, you can specify a friendly application name in the **Display Name** and **Tile Title** fields.

This is only a start. Documentation for the individual platforms provide much more detail about icons and splash screens and other application packaging issues, including submitting the application to the various stores and marketplaces.

Now that the **NoteTaker** application has reached its final state, it's time to explore various aspects of Xamarin.Forms programming in more detail.

CHAPTER 5

Principles of presentation

As you've seen, the user interface of a Xamarin.Forms application is built from instances of classes that derive from `Page`, `Layout`, and `View`. You've seen how `StackLayout` can organize views on the page and how a program can navigate among pages.

Over the next two chapters, let's explore views and layouts in more depth, beginning in this chapter with the two views devoted mostly to presentation purposes—`Label` for displaying text and `Image` for displaying bitmaps—and save the more interactive views for the next chapter.

Also introduced in this chapter is a second layout called `AbsoluteLayout`, which lets you define the coordinate positions and sizes of views on the page. This might at first seem like a particularly old-fashioned type of layout, and something that thoroughly modern programmers avoid except for graphics programming, but you'll see that `AbsoluteLayout` has a special feature that makes it particularly enticing.

The Xamarin.Forms class hierarchy

In an object-oriented programming framework like Xamarin.Forms, a class hierarchy can often reveal important inner structures—how various classes relate to each other and the properties, methods, and events they share.

You can construct such a class hierarchy by laboriously going through the online documentation and taking note of what classes derive from what other classes. Or, you can write a Xamarin.Forms program to construct a class hierarchy and display it on the phone. Such a program makes use of .NET reflection to obtain all the public classes, structures, and enumerations in the Xamarin.Forms.Core assembly and to arrange them in a tree. The **ClassHierarchy** application demonstrates this technique.

The **ClassHierarchy** project contains a class that derives from `ContentPage` named `ClassHierarchyPage`, and two additional classes named `TypeInformation` and `ClassAndSubclasses`.

The program creates one `TypeInformation` instance for every public class (and structure and enumeration) in the Xamarin.Forms.Core assembly, plus any .NET class that serves as a base class for any Xamarin.Forms class, with the exception of `Object`. (These .NET classes are `Attribute`, `Delegate`, `Enum`, `EventArgs`, `Exception`, `MulticastDelegate`, and `ValueType`.) The `TypeInformation` constructor requires only a `Type` object identifying that type but also obtains some other information:

```
namespace ClassHierarchy
{
    class TypeInformation
```

```
    {
        bool isBaseGenericType;
        Type baseGenericTypeDef;

        public TypeInformation(Type type)
        {
            this.Type = type;
            TypeInfo typeInfo = type.GetTypeInfo();
            this.BaseType = typeInfo.BaseType;

            if (this.BaseType != null)
            {
                TypeInfo baseTypeInfo = this.BaseType.GetTypeInfo();
                this.isBaseGenericType = baseTypeInfo.IsGenericType;

                if (this.isBaseGenericType)
                {
                    this.baseGenericTypeDef = baseTypeInfo.GetGenericTypeDefinition();
                }
            }
        }

        public Type Type { private set; get; }
        public Type BaseType { private set; get; }

        public bool IsDerivedDirectlyFrom(Type parentType)
        {
            if (this.BaseType != null && this.isBaseGenericType)
            {
                if (this.baseGenericTypeDef == parentType)
                {
                    return true;
                }
            }
            else if (this.BaseType == parentType)
            {
                return true;
            }
            return false;
        }
    }
}
```

A very important part of this class is the `IsDerivedDirectlyFrom` method, which will return `true` if passed an argument that is this type's base type. This determination is complicated if generic classes are involved, and that issue largely accounts for the complexity of the class.

The `ClassAndSubclasses` class is considerably shorter:

```
namespace ClassHierarchy
{
    class ClassAndSubclasses
    {
        public ClassAndSubclasses(Type parent)
```

```
        {
            this.Type = parent;
            this.Subclasses = new List<ClassAndSubclasses>();
        }

        public Type Type { private set; get; }
        public List<ClassAndSubclasses> Subclasses { private set; get; }
    }
}
```

The program creates one instance of this class for every `Type` displayed in the class hierarchy, including `Object`, so the program creates one more `ClassAndSubclasses` instance than the number of `TypeInformation` instances. The `ClassAndSubclasses` instance associated with `Object` will contain a collection of all the classes that derive directly from `Object`, and each of those `ClassAndSubclasses` instances contains a collection of all the classes that derive from that one, and so forth, for the remainder of the hierarchy tree.

Here's the `ClassHierarchyPage` class. It obtains a reference to the Xamarin.Forms `Assembly` object, and then accumulates all the public classes, structures, and enumerations in the `classList` collection defined as a field. It then checks for the necessity of including any base classes from the .NET assemblies, sorts the result, and then calls two recursive methods, `AddChildrenToParent` and `AddItemToStackLayout`:

```
class ClassHierarchyPage : ContentPage
{
    Assembly xamarinFormsAssembly;
    List<TypeInformation> classList = new List<TypeInformation>();
    StackLayout stackLayout;

    public ClassHierarchyPage()
    {
        // Get Xamarin.Forms assembly.
        xamarinFormsAssembly = typeof(View).GetTypeInfo().Assembly;

        // Loop through all the types.
        foreach (Type type in xamarinFormsAssembly.ExportedTypes)
        {
            TypeInfo typeInfo = type.GetTypeInfo();

            // Public types only but exclude interfaces
            if (typeInfo.IsPublic && !typeInfo.IsInterface)
            {
                // Add type to list.
                classList.Add(new TypeInformation(type));
            }
        }

        // Ensure that all classes have a base type in the list.
        //  (i.e., add Attribute, ValueType, Enum, EventArgs, etc.)
        int index = 0;

        // Watch out! Loops through expanding classList!
```

```
        do
        {
            // Get a child type from the list.
            TypeInformation childType = classList[index];

            if (childType.Type != typeof(Object))
            {
                bool hasBaseType = false;

                // Loop through the list looking for a base type.
                foreach (TypeInformation parentType in classList)
                {
                    if (childType.IsDerivedDirectlyFrom(parentType.Type))
                        hasBaseType = true;
                }

                // If there's no base type, add it.
                if (!hasBaseType && childType.BaseType != typeof(Object))
                {
                    classList.Add(new TypeInformation(childType.BaseType));
                }
            }
            index++;
        }
        while (index < classList.Count);

        // Now sort the list.
        classList.Sort((t1, t2) =>
        {
            return String.Compare(t1.Type.Name, t2.Type.Name);
        });

        // Start the display with System.Object.
        ClassAndSubclasses rootClass = new ClassAndSubclasses(typeof(Object));

        // Recursive method to build the hierarchy tree.
        AddChildrenToParent(rootClass, classList);

        // Create the StackLayout for displaying the list.
        stackLayout = new StackLayout
        {
            Spacing = 0
        };

        // Recursive method for adding items to StackLayout.
        AddItemToStackLayout(rootClass, 0);

        // Put the StackLayout in a ScrollView.
        this.Padding = new Thickness(5, Device.OnPlatform(20, 0, 0), 0, 0);
        this.Content = new ScrollView
        {
            Content = stackLayout
        };
    }
```

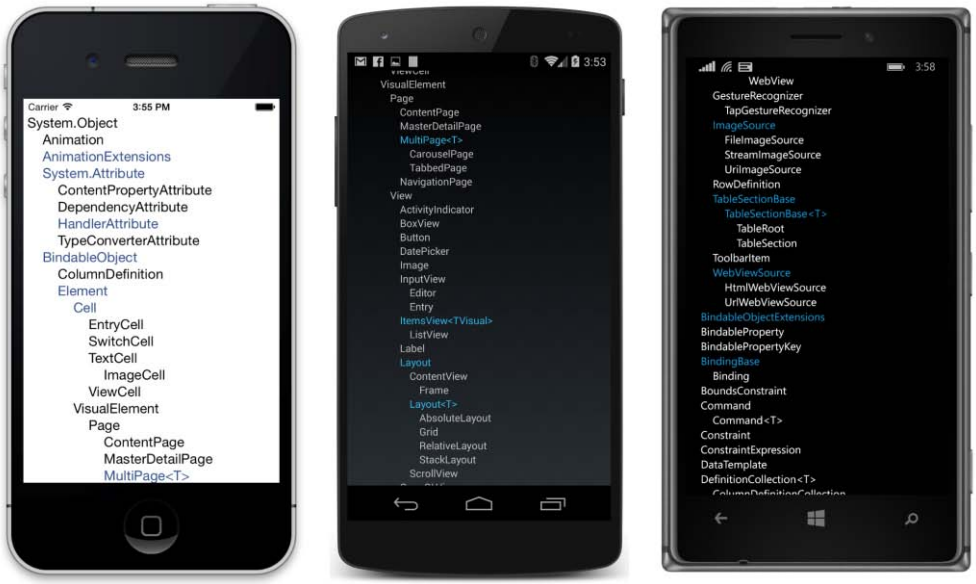
```
void AddChildrenToParent(ClassAndSubclasses parentClass,
                         List<TypeInformation> classList)
{
    foreach (TypeInformation typeInformation in classList)
    {
        if (typeInformation.IsDerivedDirectlyFrom(parentClass.Type))
        {
            ClassAndSubclasses subClass =
                new ClassAndSubclasses(typeInformation.Type);
            parentClass.Subclasses.Add(subClass);
            AddChildrenToParent(subClass, classList);
        }
    }
}

void AddItemToStackLayout(ClassAndSubclasses parentClass, int level)
{
    // If assembly is not Xamarin.Forms, display full name.
    string name = parentClass.Type.Name;
    TypeInfo typeInfo = parentClass.Type.GetTypeInfo();

    if (typeInfo.Assembly != xamarinFormsAssembly)
    {
        name = parentClass.Type.FullName;
    }

    // If generic, display angle brackets and parameters.
    if (typeInfo.IsGenericType)
    {
        Type[] parameters = typeInfo.GenericTypeParameters;
        name = name.Substring(0, name.Length - 2);
        name += "<";

        for (int i = 0; i < parameters.Length; i++)
        {
            name += parameters[i].Name;
            if (i < parameters.Length - 1)
                name += ", ";
        }
        name += ">";
    }

    // Create Label and add to StackLayout
    Label label = new Label
    {
        Text = String.Format("{0}{1}", new string(' ', 4 * level), name),
        TextColor = parentClass.Type.GetTypeInfo().IsAbstract ?
                        Color.Accent : Color.Default
    };

    stackLayout.Children.Add(label);

    // Now display nested types.
```

```
            foreach (ClassAndSubclasses subclass in parentClass.Subclasses)
                AddItemToStackLayout(subclass, level + 1);
        }
    }
```

The recursive `AddChildrenToParent` method assembles the linked list of `ClassAndSubclasses` instances from the flat `classList` collection.

The `AddItemToStackLayout` is also recursive because it is responsible for adding the `ClassesAndSubclasses` linked list to the `StackLayout` object by creating a `Label` view for each class with a little blank space at the beginning for the proper indentation.

The method displays the Xamarin.Forms types with just the class names but the .NET types with the fully-qualified name to distinguish them. The method uses the platform highlight color for classes that are not instantiable either because they are abstract or static:



Overall, you'll see that the Xamarin.Forms visual elements have the following general hierarchy:

```
System.Object
    BindableObject
        Element
            VisualElement
                View
                    …
                    Layout
                        …
                        Layout<T>
```

```
                                        ...
                    Page
                        ...
```

Aside from `Object`, all these classes are implemented in the Xamarin.Forms.Core.dll assembly and associated with a namespace of `Xamarin.Forms`.

As the name of the `BindableObject` class implies, this class supports data binding—the linking of two properties of two objects so that they maintain the same value. The class provides for special property definitions called `BindableProperty` objects, and implements the .NET `INotifyPropertyChanged` interface. You've already seen some data binding, and more will be demonstrated in the next chapter.

User interface objects in Xamarin.Forms are often arranged on the page in a parent-child hierarchy, and the `Element` class includes support for parent and child relationships.

`VisualElement` is an exceptionally important class in Xamarin.Forms. A visual element is anything in Xamarin.Forms that occupies an area on the screen. The `VisualElement` class defines 25 properties related to size, location, background color, and other visual and functional characteristics such as `IsEnabled` and `IsVisual`.

In Xamarin.Forms the word *view* is often used to refer to individual visual objects such as buttons, sliders, and text-entry boxes, but you can see that the `View` class is parent to the layout classes as well. Interestingly, `View` only adds a couple of public members to what it inherits from `VisualElement`. Those include `HorizontalOptions` and `VerticalOptions`, which makes sense because those properties don't apply to pages. Another property defined by `View` is `GestureRecognizers`, which consolidates the support of touch input in Xamarin.Forms.

The descendants of `Layout` are capable of having children views. A child view appears on the screen visually within the boundaries of its parent. Classes that derive from `Layout` can have only one child of type `View`, but the generic `Layout<T>` class defines a `Children` property, which is a collection of multiple child views, including other layouts. You've already seen the `StackLayout`, which arranges its children in a horizontal or vertical stack. Later in this chapter you'll see `AbsoluteLayout`. Although the `Layout` class derives from `View`, layouts are so important in Xamarin.Forms that they often considered a category in themselves.

Although **ClassHierarchy** lists all the public classes, structures, and enumerations defined by Xamarin.Forms, it does not list interfaces. Those are important as well but you'll just have to explore them on your own. (Or enhance the program to list them.)

Pixels, points, dps, DIPs, and DIUs

The **ClassHierarchy** program uses a `Label` with a default `Font` object for displaying text. Yet each platform displays a different number of lines on the screen.

Is the quantity of text displayable on the screen something that a Xamarin.Forms application can anticipate or control? And even if it's possible, is it a proper programming practice? Should an application adjust font sizes to achieve a particular text density on the screen?

When programming a Xamarin.Forms application, it's usually best not to get too close to the actual numeric dimensions of visual objects. It's preferable to trust Xamarin.Forms and the three platforms to make the best default choices and lay out the views.

However, there are times when the programmer needs to know something about the size of particular visual objects, and the size of the screen on which they appear. Video displays consist of a rectangular array of pixels, and although these pixels are essential for rendering text and graphics, pixels cause major headaches for programmers attempting to write applications that look roughly the same on a variety of devices.

Solutions to the problem of working with pixels on mobile platforms originated on desktop platforms, so it's illuminating to begin there. Desktop video displays have a wide range of pixel dimensions, from the nearly obsolete 640 × 480 on up into the thousands. The aspect ratio of 4:3 was once standard for computer displays—and for movies and television as well—but the high-definition aspect ratio of 16:9 (or the similar 16:10) is now more common.

Desktop video displays also have a physical dimension usually measured along the diagonal in inches or centimeters. The pixel dimension combined with the physical dimension allows calculating the video display's *resolution* or *pixel density* in dots per inch (DPI), sometimes also referred to as pixels per inch (PPI). The display resolution can also be measured as a *dot pitch*, which is the distance between pixel centers, usually measured in millimeters.

For example, you can use the Pythagorean Theorem to calculate that an ancient 800 × 600 display contains 1000 pixels diagonally. If this monitor has a 13" diagonal, that's a pixel density of 77 DPI, or a dot pitch of 0.33 millimeters. However, the 13" screen on a modern laptop might have pixel dimensions of 2560 × 1600, which is a pixel density of about 230 DPI, or a dot pitch of about 0.11 millimeters. A 100-pixel square object on this screen is 1/3rd the size of the same object on the older screen.

Programmers should have a fighting chance when attempting to size visual elements correctly. For this reason, both Apple and Microsoft devised systems for desktop computing that allow programmers to work with the video display in some form of device-independent units rather than pixels. Most of the dimensions that a programmer encounters and specifies are in these device-independent units. It is the responsibility of the operating system to convert back and forth between these units and pixels.

In the Apple world, desktop video displays were traditionally assumed to have a resolution of 72 units to the inch. This number comes from typography: In classical typography, measurements are in units of *points*, and there are approximately 72 points to the inch. (In digital typography the point has been standardized to exactly 1/72nd inch.) By working with points rather than pixels, the programmer has an intuitive sense of the relationship between numeric sizes and the area that visual objects occupy on the screen.

In the Microsoft Windows world, a similar technique was developed called *device-independent pixels* (DIPs) or *device-independent units* (DIUs). To the Microsoft programmer, desktop video displays are assumed to have a resolution of 96 DIUs, which is exactly 1/3 higher than 72 DPI.

Phones, however, have some different rules: The pixel densities achieved on modern phones are typically much higher than desktop displays. This higher pixel density allows text and other visual objects to shrink much more in size before becoming illegible.

Phones are also typically held much closer to the user's face than a desktop or laptop screen. This difference also implies that visual objects on the phone can be much smaller than comparable objects on desktop or laptop screens. Because the physical dimensions of the phone are much smaller than desktop displays, shrinking down visual objects is very desirable because it allows much more to fit on the screen.

Apple continues to refer to the device-independent units on the iPhone as *points*, but for all Apple's high-density displays—which Apple refers to by the brand name *Retina*—there are two pixels to the point. This is true for the iPhone, iPad, and MacBook Pro.

For example, the 640 × 960 pixel dimension of the 3.5" screen of the iPhone 4 has an actual pixel density of about 320 DPI. To the programmer, the screen appears to have a dimension of 320 × 480 points. The iPhone 3 actually did have a pixel dimension of 320 × 480, and points equaled pixels, so to the programmer the displays of the iPhone3 and iPhone4 appear to be the same size. But this size implies a conversion factor of 160 points to the inch rather than 72.

The iPhone 5 has a 4" screen, but the pixel dimension is 640 × 1136, and the pixel density is about the same as the iPhone 4. To the programmer, this screen has a size of 320 × 768 points.

Android does something quite similar: Android devices have a wide variety of sizes and pixel dimensions, but the Android programmer always works in units of *density-independent pixels (dps)*. The relationship between pixels and dps is set assuming 160 dps to the inch, which means that Apple and Android device-independent units are very similar.

Microsoft took a different approach with the Windows Phone, however. Windows Phone 7 devices have a uniform pixel dimension of 480 × 800, which is often referred to as WVGA (Wide Video Graphics Array). Windows Phone 7 programmers work with this display in units of pixels, which means that the Windows Phone is implicitly assuming a pixel density of about 240 DPI, which is 1.5 times the assumed pixel density of iPhone and Android devices.

With Windows Phone 8, several larger screen sizes are allowed: 768 × 1280 (WXGA or Wide Extended Graphics Array), 720 × 1280 (referred to using high-definition television lingo as 720p), and 1080 × 1920 (called 1080p).

For these additional display sizes, programmers work in device-independent units. An internal scaling factor translates between pixels and device-independent units so that the width of the screen in portrait mode always appears to be 480 pixels wide. The scaling factors are 1.6 (for WXGA), 1.5 (720p), and 2.25 (1080p).

Xamarin.Forms has a philosophy of using the conventions of the underlying platforms as much as possible. In accordance with this philosophy, the Xamarin.Forms programmer works with sizes of visual elements set by each particular platform. Xamarin.Forms programmers can generally treat the phone display in a device-independent manner, but a little differently for the three platforms: For iOS and Android devices, the programmer can assume that the screen has a density of 160 device-independent units to the inch; for Windows Phone, the density is 240 units to the inch. If it's desirable to size visual objects so that they appear about the same physical size on all three platforms, dimensions on the Windows Phone should be about 150% larger than dimensions on the iPhone and Android.

The `VisualElement` class defines two properties named `Width` and `Height` that provide the rendered dimension of views, layouts, and pages in these device-independent units. However, the initial settings of `Width` and `Height` are "mock" values of –1. The values of these properties only become valid when the layout system has positioned and sized everything on the page. Also, keep in mind that the default `HorizontalOptions` or `VerticalOptions` settings of `Fill` often cause a view to occupy more space than it would otherwise, and the `Width` and `Height` properties reflect this extra space. The `Width` and `Height` properties are consistent with the area colored by the view's `BackgroundColor`.

`VisualElement` defines an event named `SizeChanged` that is fired whenever the `Width` and `Height` properties of the visual element change. You can attach a `SizeChanged` handler to any visual object on the page, including the page itself. The **WhatSize** program demonstrates how to obtain this size and display it.

```
class WhatSizePage : ContentPage
{
    Label label;

    public WhatSizePage()
    {
        label = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        this.Content = label;
        this.SizeChanged += OnPageSizeChanged;
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        label.Text = String.Format("{0} ¥u00D7 {1}", this.Width, this.Height);
    }
}
```

The Unicode character in the `String.Format` call is a times symbol.

A protected virtual method named `OnSizeAllocated` also indicates when the visual element is

assigned a size. You can override this method in your `ContentPage` derivative rather than handling the `SizeChanged` event, but `OnSizeAllocated` is sometimes called when the size isn't actually changing.

Notice that the **WhatSize** program does not create a new `Label` in its `SizeChanged` event handler. It certainly could create a new `Label` and set it as the new content of the page (in which case the previous `Label` would become unreferenced and hence eligible for garbage collection), but it's unnecessary and wasteful. The program creates only one `Label` in its constructor and then just sets the `Text` property to indicate the page's new size:

For the record, these are the sources of the screens in these three images:

- The **iPhone Retina (3.5-inch) / iOS 7.1** simulator, with pixel dimensions of 640 × 960.
- An LG Nexus 5 with a screen size of 1080 × 1920 pixels.
- A Nokia Lumia 925 with a screen size of 768 × 1280 pixels.

Notice that the vertical size perceived by the program on the Android does not include the area occupied by the status bar or bottom buttons; the vertical size on the Windows Phone does not include the area occupied by the status bar.

If the application is set to respond to device orientation changes, the page size reflects the orientation of the phone. This means that the numbers displayed by **WhatSize** change as you rotate the phone or emulator side to side. When these three phones are turned sideways, they display the following dimensions:

- iPhone: 480 × 320.

- Android: 598 × 335. The 598-pixel width excludes the area for the buttons; the 335-pixel height excludes the status bar, which always appears above the page.
- Windows Phone: 728 × 480. The 728-pixel width excludes the area for the status bar, which appears in the same place but which rotates its icons to reflect the new orientation.

Applications need to be enabled to respond to device orientation changes. The standard Xamarin.Forms project templates for iOS and Android enable device orientation changes automatically. For Windows Phone, you need to add this statement to the Windows Phone `MainPage` constructor right after `InitializeComponent`:

```
this.SupportedOrientations = SupportedPageOrientation.PortraitOrLandscape;
```

Monitoring size changes is the only way a Xamarin.Forms application can detect orientation changes without obtaining platform-specific information. Is the width greater than the height? That's landscape. Otherwise, it's portrait.

Estimated font sizes

The various static methods of `Font` let you specify font size in two different ways: as a member of the `NamedSize` enumeration (`Large`, `Medium`, `Small`, or `Micro`) or with a numeric size. This numeric font size is in the same device-independent units used through Xamarin.Forms. As you've seen, you can calculate device-independent units based on the platform resolution:

- iOS: 160 dots per inch
- Android: 160 dots per inch
- Windows Phone: 240 dots per inch

For example, suppose you want to use a 12-point font in your program. The first thing you should know is that while a 12-point font might be a comfortable size for printed material or a desktop screen, on a phone it's quite large. But let's continue.

There are 72 points to the inch, so a 12-point font is 1/6th inch. Multiply by the DPI resolution. That's about 27 device-independent units on iOS and Android and 40 device-independent units on Windows Phone.

Let's write a little program called **FontSizes** that displays text using the `NamedSize` values and some numeric point sizes, converted to device independent units using the device resolution:

```
class FontSizesPage : ContentPage
{
    public FontSizesPage()
    {
        BackgroundColor = Color.White;

        StackLayout stackLayout = new StackLayout();
```

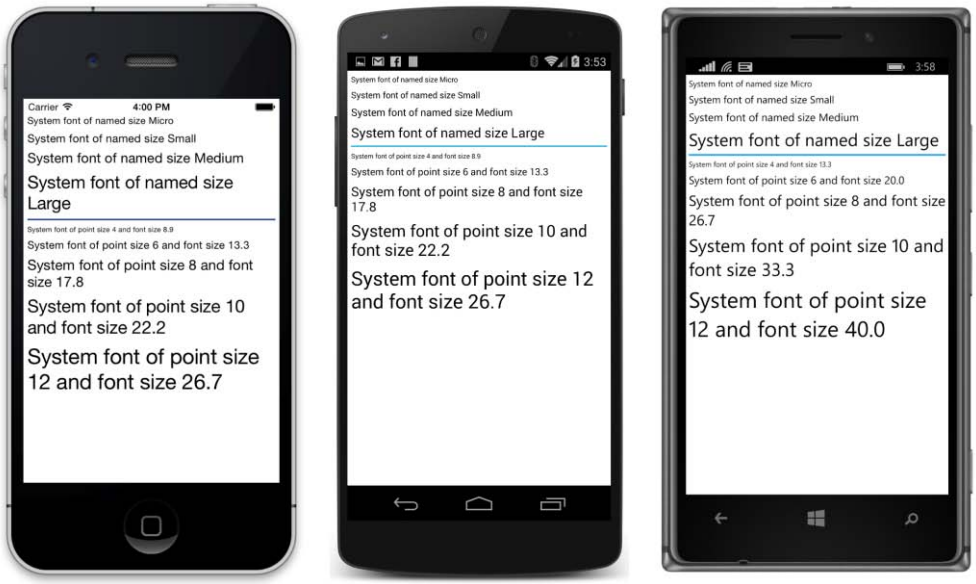
```
        // Add NamedSize fonts to StackLayout.
        foreach (NamedSize namedSize in Enum.GetValues(typeof(NamedSize)))
        {
            stackLayout.Children.Add(
                new Label
                {
                    Text = String.Format("System font of named size {0}",
                                         namedSize),
                    Font = Font.SystemFontOfSize(namedSize),
                    TextColor = Color.Black
                });
        }

        // Resolution in device-independent units per inch.
        double resolution = Device.OnPlatform(160, 160, 240);

        // Draw horizontal separator line.
        stackLayout.Children.Add(
            new BoxView
            {
                Color = Color.Accent,
                HeightRequest = resolution / 80
            });

        // Add numeric sized fonts to StackLayout.
        int[] ptSizes = { 4, 6, 8, 10, 12 };

        foreach (double ptSize in ptSizes)
        {
            double fontSize = resolution * ptSize / 72;

            stackLayout.Children.Add(
                new Label
                {
                    Text = String.Format("System font of point size " +
                                         "{0} and font size {1:F1}",
                                         ptSize, fontSize),
                    Font = Font.SystemFontOfSize(fontSize),
                    TextColor = Color.Black
                });
        }

        this.Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);
        this.Content = stackLayout;
    }
}
```

To facilitate comparisons among the three screens, the backgrounds have been uniformly set to white and the labels to black.

Notice the `BoxView` inserted into the `StackLayout` between the two `foreach` statements. The `HeightRequest` setting gives it a device-independent height of approximately 1/80th inch, and it

resembles a horizontal rule.

The resultant font sizes are perhaps not as consistent between the three platforms as you might prefer, but they provide a rough idea what you can expect:



Interestingly, the numeric sizes calculated using the device resolution seem a bit more consistent among the platforms than the named sizes, at least on these devices.

You might need to fit a block of text to a particular rectangular area. Two numbers can help with this job:

The first is line spacing. This is the vertical height of a `Label` view per line of text. It is roughly related to the font size specified in the `Font.SystemFontOfSize` method as follows:

- iOS: *LineSpacing* = 1.2 * *FontSize*
- Android: *LineSpacing* = 1.2 * *FontSize*
- Windows Phone: *LineSpacing* = 1.3 * *FontSize*

The second helpful number is average character width. This is about half of the font size, regardless of the platform:

- *AverageCharacterWidth* = 0.5 * *FontSize*

For example, suppose you want to fit a text string containing 80 characters in a width of 320 units. You'd like the font size to be as large as possible. Divide the width (320) by half the number of characters (40), and you get a font size of 8 that you can use in the `Font.SystemFontOfSize` method. For text that's somewhat indeterminate and can't be tested beforehand, you might want to

make this calculation a little more conservative to avoid surprises. (Or, you can set a `SizeChanged` handler on a `Label` and adjust the font size dynamically, but that process will generate another `SizeChanged` event so the process can be tricky.)

The following program uses both line spacing and average character width to fit a paragraph of text on the page—or rather, on the page minus the area at the top of the iPhone occupied by the status bar. To make the exclusion of the status bar a bit easier in this program, the program uses a `ContentView`.

`ContentView` derives from `Layout` but only adds a `Content` property to what it inherits from `Layout`. `ContentView` is the base class to `Frame` and otherwise seems to serve no apparent purpose—except when it comes in really handy.

As you might have noticed, Xamarin.Forms has no concept of a *margin*, which traditionally is similar to padding except that padding is inside a view and a part of the view, while a margin is outside the view and actually part of the parent's view.

Xamarin.Forms doesn't have a margin property, but you can simulate it very easily with a `ContentView`. If you find a need to set a margin on a view, put the view in a `ContentView` and set the `Padding` property on the `ContentView`. `ContentView` inherits a `Padding` property from `Layout`.

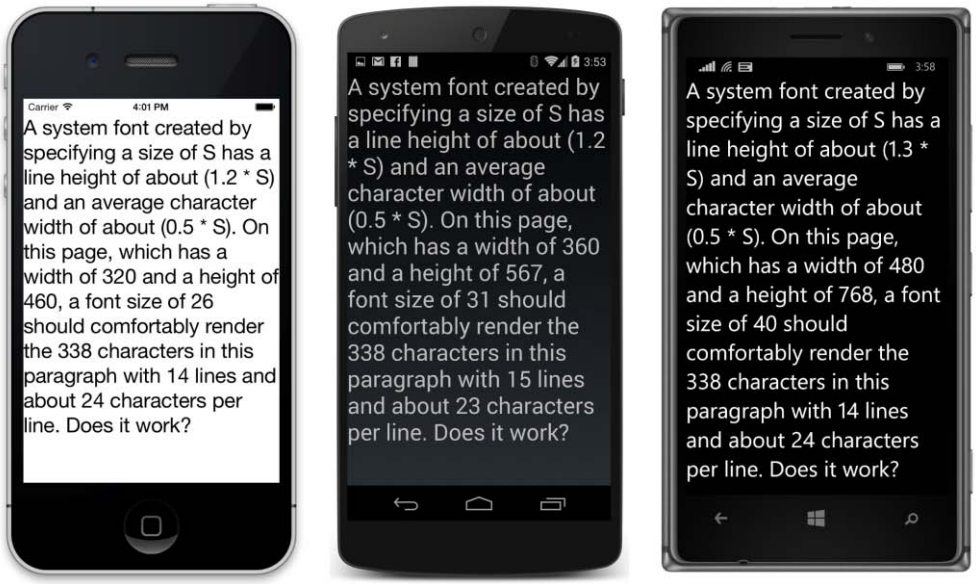
The **EstimatedFontSize** program uses `ContentView` in a slightly different manner: It sets the customary `Padding` on the page to avoid the iOS status bar, but then sets a `ContentView` as the content of that page. Hence, this `ContentView` is the same size as the page but excluding the iOS status bar.

It is on this `ContentView` that the `SizeChanged` event is attached, and it is the size of this `ContentView` that is used to calculate the text font size. The calculation is described in comments:

```
class EstimatedFontSizePage : ContentPage
{
    Label label;

    public EstimatedFontSizePage()
    {
        // Create the Label.
        label = new Label();

        // Put the Label in a ContentView.
        ContentView contentView = new ContentView
        {
            Content = label
        };

        // Monitor the ContentView size changes!
        contentView.SizeChanged += OnContentViewSizeChanged;

        // A little padding for iOS.
        this.Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
```

```
        this.Content = contentView;
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        string text =
            "A system font created by specifying a size of S " +
            "has a line height of about ({0:F1} * S) and an " +
            "average character width of about ({1:F1} * S). " +
            "On this page, which has a width of {2:F0} and a " +
            "height of {3:F0}, a font size of ?1 should " +
            "comfortably render the ??2 characters in this " +
            "paragraph with ?3 lines and about ?4 characters " +
            "per line. Does it work?";

        // Get View whose size is changing.
        View view = (View)sender;

        // Two values as multiples of font size
        double lineHeight = Device.OnPlatform(1.2, 1.2, 1.3);
        double charWidth = 0.5;

        // Format the text and get its length
        text = String.Format(text, lineHeight, charWidth, view.Width, view.Height);
        int charCount = text.Length;

        // Because:
        //   lineCount = view.Height / (lineHeight * fontSize)
        //   charsPerLine = view.Width / (charWidth * fontSize)
        //   charCount = lineCount * charsPerLine
        // Hence, solving for fontSize:
        int fontSize = (int)Math.Sqrt(view.Width * view.Height /
                                      (charCount * lineHeight * charWidth));

        // Now these values can be calculated.
        int lineCount = (int)(view.Height / (lineHeight * fontSize));
        int charsPerLine = (int)(view.Width / (charWidth * fontSize));

        // Replace the placeholders with the values.
        text = text.Replace("?1", fontSize.ToString());
        text = text.Replace("??2", charCount.ToString());
        text = text.Replace("?3", lineCount.ToString());
        text = text.Replace("?4", charsPerLine.ToString());

        // Set the Label properties.
        label.Text = text;
        label.Font = Font.SystemFontOfSize(fontSize);
    }
}
```

The goal is to make the text as large as possible without the text spilling off the page:



Not bad. Not bad at all. The text actually displays on the iPhone in 13 lines, and the Android in 14 lines, but the technique seems sound.

Programmer's first clock app

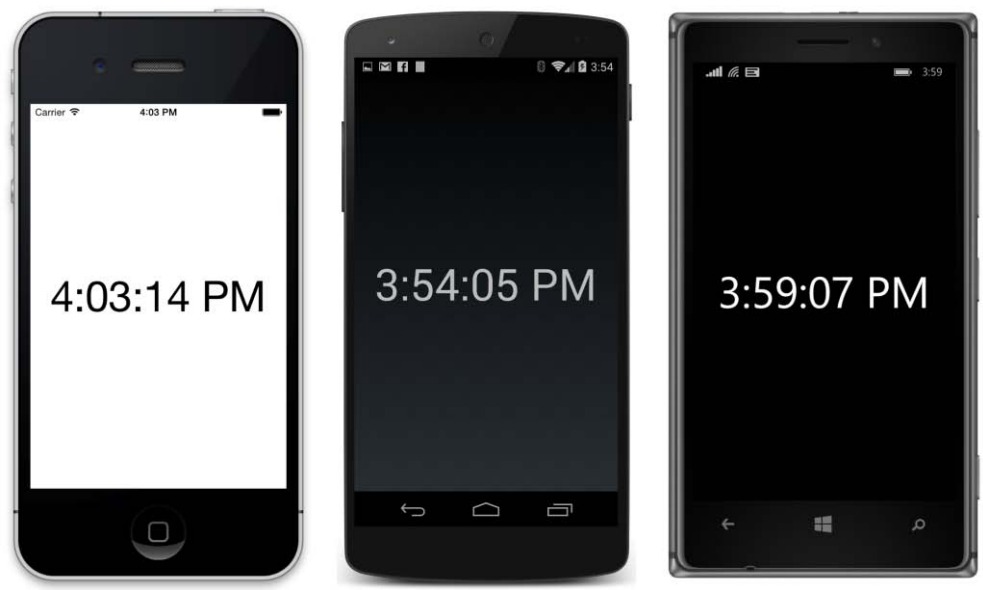
The `Device` class includes a static `StartTimer` method that lets you set a timer that fires a periodic event. The availability of a timer event means that a clock application is possible, even if it only displays the time in text:

```
class DigitalClockPage : ContentPage
{
    Label clockLabel;

    public DigitalClockPage()
    {
        // Create the centered Label for the clock display.
        clockLabel = new Label
        {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        this.Content = clockLabel;

        // Set a SizeChanged event on the page.
        this.SizeChanged += OnPageSizeChanged;

        // Start the timer going.
```

```
        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerTick);
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // Scale the font size to the page width
        //      (based on 11 characters in the displayed string).
        if (this.Width > 0)
            clockLabel.Font = Font.SystemFontOfSize(this.Width / 6);
    }

    bool OnTimerTick()
    {
        // Set the Text property of the Label.
        clockLabel.Text = DateTime.Now.ToString("h:mm:ss tt");
        return true;
    }
}
```

The program creates a `Label` for displaying the time and then sets two events: the `SizeChanged` event on the page for changing the font size, and the `Device.StartTimer` event for one-second intervals. Both event handlers simply change a property of the `Label`. The `Device.StartTimer` event must return `true` to continue the timer and `false` to stop it.

The `OnTimerTick` method specifies a custom formatting string for `DateTime` that results in 10 or 11 characters, but two of those are capital letters, and those are wider than average characters. The `SizeChanged` handler implicitly assumes that 12 characters are displayed by setting the font size to 1/6 of the page width:



Of course, the text is much larger in landscape mode. Unfortunately the screenshots for this book

are designed only for portrait mode, so you'll need to turn this book sideways to see what the program looks like in landscape:

Of course this one-second timer doesn't tick exactly at the beginning of every second, so the displayed time might not precisely agree with other time displays on the same device. You can make it more accurate by setting a faster timer tick. Performance won't be impacted much because the display still changes only once per second and won't require a new layout cycle until then.

Image and bitmaps

For the purposes of presentation, text is essential but pictures are… well, pictures are usually essential as well.

A view named `Image` does for bitmaps what `Label` does for text. A bitmap displayed by `Image` can be obtained from a website (or otherwise accessible via a URL), or it can be bound into the program's PCL as a resource, or it can be content of the individual platform projects or an SAP, or it can be otherwise accessible through a .NET `Stream` object.

`Image` defines a `Source` property that you set to an object of type `ImageSource`, which defines four static methods for referencing a bitmap. These are:

- `ImageSource.FromUri` for accessing a bitmap over the web.
- `ImageSource.FromResource` for a bitmap stored as a resource in the application PCL.
- `ImageSource.FromFile` for a bitmap stored as content in the platform projects or an SAP.

- `ImageSource.FromStream` for loading a bitmap using a .NET `Stream` object.

`ImageSource` also has three descendant classes, named `UriImageSource`, `FileImageSource`, and `StreamImageSource`, that you can use instead of the static methods, but generally the static methods provide a little more support. The `ImageSource.FromResource` method uses code similar to the **BlackCat** program in Chapter 2 to obtain a `Stream` object for accessing a resource in the PCL.

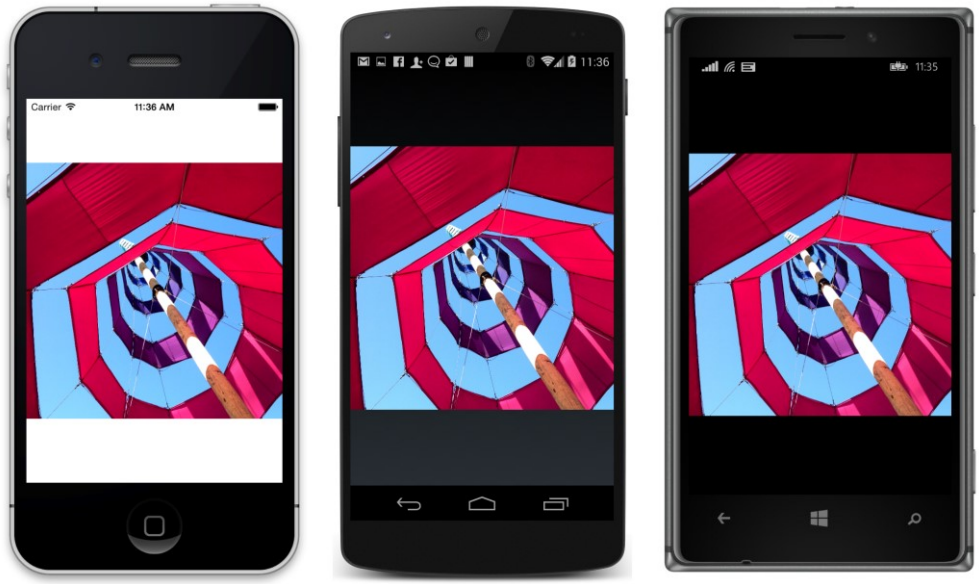
Here's a program that uses `ImageSource.FromUri` to access a bitmap from Xamarin's website:

```
class BitmapFromWebsitePage : ContentPage
{
    public BitmapFromWebsitePage()
    {
        string uri = "http://developer.xamarin.com/demo/IMG_1415.JPG";

        this.Content = new Image
        {
            Source = ImageSource.FromUri(new Uri(uri))
        };
    }
}
```

Even this tiny program can be simplified. `ImageSource` defines an implicit conversion from `string` or `Uri`, so you can set the URL directly to the `Source` property:

```
class BitmapFromWebsitePage : ContentPage
{
    public BitmapFromWebsitePage()
    {
        this.Content = new Image
        {
            Source = "http://developer.xamarin.com/demo/IMG_1415.JPG"
        };
    }
}
```

By default, the bitmap displayed by the `Image` view is stretched to the size of the page but respecting the bitmap's aspect ratio:



This precise fit occurs regardless of whether the bitmap size is less than or greater than the page size. This bitmap happens to be 4,096 pixels square, but you can download a much smaller file by specifying the `uri` variable like so:

```
string uri = "http://developer.xamarin.com/demo/IMG_1415.JPG?width=100";
```

The image is still stretched to fit the area.

Consequently, as you turn your phone or emulator between portrait and landscape mode, a rendered bitmap can change size, and you'll see some blank space at the top and bottom, or the left and right, where the bitmap doesn't reach. You can color that space by using the `BackgroundColor` property that `Image` inherits from `VisualElement`.

`Image` defines an `Aspect` property that controls how the bitmap is rendered. The default setting is the enumeration member `Aspect.AspectFit`, meaning that the bitmap fits into its parent's boundaries while preserving the bitmap's aspect ratio.

Try this:

```
class BitmapFromWebsitePage : ContentPage
{
    public BitmapFromWebsitePage()
    {
        string uri = "http://developer.xamarin.com/demo/IMG_1415.JPG";

        this.Content = new Image
        {
            Source = ImageSource.FromUri(new Uri(uri)),
            Aspect = Aspect.Fill
        };
```

```
    }
}
```

Now the bitmap completely fills the page at the expense of distorting the image.

The third option is `Aspect.AspectFill`. With this option the bitmap completely fills the page but maintains the bitmap's aspect ratio at the same time. The only way this is possible is by cropping part of the image, and you'll see that the image is cropped on either the top and bottom, or the left and right, so the result is centered.

If the bitmap's pixel size is less than the size of the page in device-independent units, and if you set `HorizontalOptions` or `VerticalOptions` to something other than `Fill`, or if you put the `Image` in a `StackLayout`, the bitmap is displayed in its pixel size on iOS and Android, and the pixel size interpreted as device-independent units on Windows Phone.

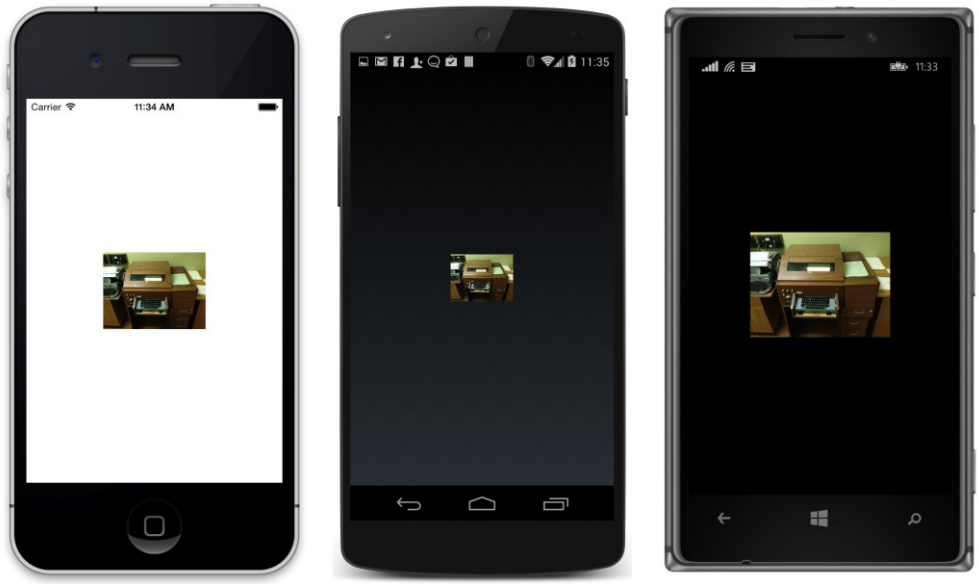
The following program, named **BitmapFromResource**, is similar to the **BlackCat** program in that it contains a folder in the PCL project and a file in that folder. This folder is named **Images**, and it contains two bitmaps named ModernUserInterface.jpg (a very large bitmap) and ModernUserInterface-256.jpg (the same picture but with a 256-pixel width).

Don't forget this crucial step: In the **Properties** setting for the two bitmap files, set the **Build Action** to **EmbeddedResource**.

The `BitmapFromResourcePage` class loads the smaller of the two bitmaps using the resource ID, which is the assembly name, followed by a period, followed by the folder, followed by another period, and the filename:

```
class BitmapFromResourcePage : ContentPage
{
    public BitmapFromResourcePage()
    {
        string resource = "BitmapFromResource.Images.IMG_1415_256.JPG";

        this.Content = new Image
        {
            Source = ImageSource.FromResource(resource),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
    }
}
```

Because the `Image` is centered on the page, the bitmap is displayed in its actual size, or the actual size interpreted as device-independent units on Windows Phone:



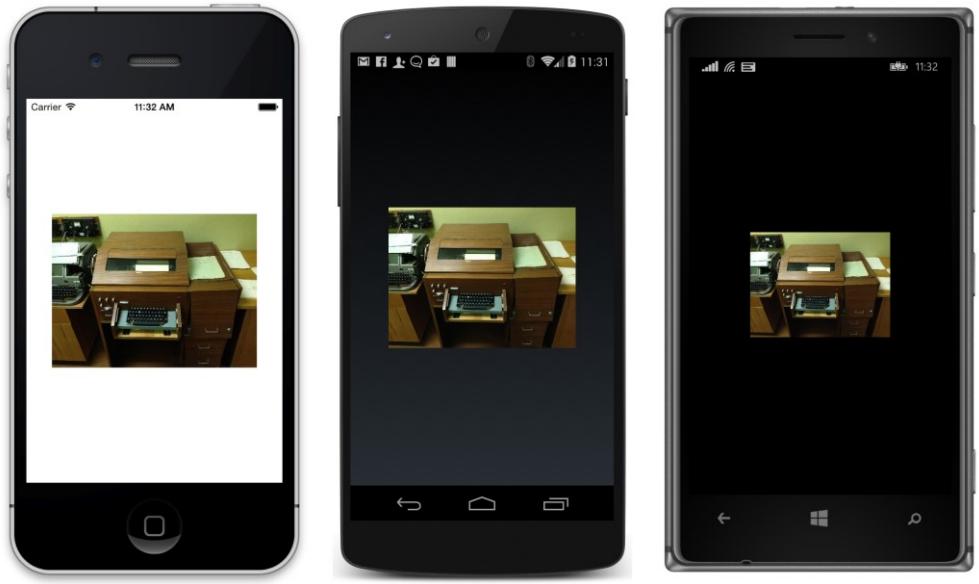
If you substitute the larger bitmap, you'll discover that it is *not* displayed in its actual pixel width (which would clearly overrun the boundaries of the screen) but instead is sized so that it fills the full width of the page—or the full height if you turn the phone into landscape mode.

Here's another option: If you set `HorizontalOptions` and `VerticalOptions` not equal to `Full`, you can also set the `WidthRequest` or `HeightRequest` to an explicit dimension smaller than the bitmap size in device-independent units. You only need to set one of these two properties because the other dimension is implied by the bitmap's aspect ratio. You can specify both `WidthRequest` and `HeightRequest` along with setting `Aspect` to `AspectFill` to stretch the bitmap to any arbitrary size.

Here's an example of that technique using the larger image:

```
class BitmapFromResourcePage : ContentPage
{
    public BitmapFromResourcePage()
    {
        string resource = "BitmapFromResource.Images.ModernUserInterface.jpg";

        this.Content = new Image
        {
            Source = ImageSource.FromResource(resource),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            WidthRequest = 256
        };
    }
}
```

Here's the result:



The image is larger than the previous code on the iPhone and Android because the `WidthRequest` is in device-independent units, but the image is the same size on Windows Phone because the previous program displays the image in device-independent units.

The `ImageSource.FromFile` method can fetch bitmaps from the individual platform projects. Here's a program that displays an icon file stored with each platform. Generally the names and folders are different for each platform, so you'll generally be using `Device.OnPlatform` in connection with `ImageSource.FromFile`:

```
class BitmapFromPlatformsPage : ContentPage
{
    public BitmapFromPlatformsPage()
    {
        this.Content = new Image
        {
            Source = ImageSource.FromFile(
                        Device.OnPlatform("Icon.png",
                                          "Icon.png",
                                          "Assets/ApplicationIcon.png"))
        };
    }
}
```

You already know how to use `ImageSource.FromFile` because it's consistent with accessing bitmaps for `ToolbarItem`. The **Build Action** for the bitmaps must be set as follows:

- iOS: **BundleResource**
- Android: **AndroidResource**
- Windows Phone: **Content**

Absolute layout and attached bindable properties

Here's the Xamarin.Forms class hierarchy showing all the classes that derive from `Layout`:

```
System.Object
    BindableObject
        Element
            VisualElement
                View
                    Layout
                        ContentView
                            Frame
                        ScrollView
                        Layout<T>
                            AbsoluteLayout
                            Grid
                            RelativeLayout
                            StackLayout
```

You've already seen `ContentView`, `Frame`, and `ScrollView` (all of which have a `Content` property that you can set to one child) and `StackLayout`, which inherits a `Children` property from `Layout<T>`. The `Grid` and `RelativeLayout` implement rather more complex layout models; `Grid` is explored in the next chapter, and `RelativeLayout` in a later chapter not included in this Preview Edition.

At first glance, the `AbsoluteLayout` class implements a rather primitive layout model—one that harkens back to the not-so-good old days of those early graphical user interfaces that required the programmer to individually size and position every element on the screen.

With `AbsoluteLayout`, most of the rules about layout that you've learned so far no longer apply. The `HorizontalOptions` and `VerticalOptions` properties that are so important when a `View` is the child of a `ContentPage` or `StackLayout` have *absolutely no effect* when a `View` is a child of an `AbsoluteLayout`. A program must instead assign to each child of an `AbsoluteLayout` a specific location. The child can also be assigned a specific size or allowed to size itself.

You can add a child view to the `Children` collection of an `AbsoluteLayout` the same way as with `StackLayout`:

```
absoluteLayout.Children.Add(child);
```

But you must also indicate the location and size of the child view by using a `Rectangle` value. You can create a `Rectangle` value with a constructor that accepts `Point` and `Size` values, or with `x`, `y`, `width`, and `height` arguments, for example:

```
Rectangle rect = new Rectangle(x, y, width, height);
```

The `x` and `y` values indicate the position of the upper-left corner of the child view relative to the upper-left corner of the `AbsoluteLayout` parent.

However, the code that you use to position and size a child view within the `AbsoluteLayout` is apt to seem a bit peculiar:

```
AbsoluteLayout.SetLayoutBounds(child, rect);
```

Does that statement look a little odd to you? That's not an instance of `AbsoluteLayout` on which you're making a `SetLayoutBounds` call. No, that's a static method, and you call `AbsoluteLayout.SetLayoutBounds` either before or after you add the child to the `AbsoluteLayout`. Indeed, because it's a static method, you can even call the method before the `AbsoluteLayout` has even been instantiated! A particular instance of `AbsoluteLayout` is not involved at all in this `SetLayoutBounds` method.

So you might wonder: What does it do? And how does it work?

That `AbsoluteLayout.SetLayoutBounds` call is equivalent to the following call on the child view object:

```
child.SetValue (AbsoluteLayout.LayoutBoundsProperty, rect);
```

In fact, this is how `AbsoluteLayout` itself defines the `SetLayoutBounds` static method.

You'll recall that `SetValue` is defined by `BindableObject` and used to implement bindable properties. Just offhand, `AbsoluteLayout.LayoutBoundsProperty` certainly appears to be a bindable property, and that is so. However, it is a very special type of bindable property called an *attached bindable property*.

Attached bindable properties are defined by one class—in this case `AbsoluteLayout`—but set on another object, in this case a child of the `AbsoluteLayout`. The property is sometimes said to be *attached* on the child; hence the name. The child of the `AbsoluteLayout` is ignorant of the purpose of the attached bindable property passed to the `SetValue` method and makes no use of that value in its own internal logic, and `SetValue` simply saves the values in a dictionary maintained by `BindableObject` within the child. The `SetValue` method simply attaches this value to the child to be used at some point by the parent—the `AbsoluteLayout`.

When the `AbsoluteLayout` is laying out its children, it can interrogate the value of this property on each child by calling the `AbsoluteLayout.GetLayoutBounds` static method on the child, which in turn calls `GetValue` on the child with the `AbsoluteLayout.LayoutBoundsProperty` attached bindable property, which fetches the `Rectangle` value from the dictionary stored within the child.

Attached bindable properties are a general-purpose mechanism for properties defined by one class to be stored in instances of another class, but attached bindable properties are used mostly with layout classes. As you'll see in the next chapter, `Grid` defines attached bindable properties to specify the row and column of each child.

It can be difficult for a program to determine a good width and height to assign to a child of an `AbsoluteLayout`, but often you don't need to. If you specify the static read-only property `AbsoluteLayout.AutoSize` for the `width` and `height` arguments in the `Rectangle` you pass to `AbsoluteLayout.SetLayoutBounds`, the child will be sized based on a width and height it calculates for itself.

`AbsoluteLayout` redefines its `Children` property to provide a couple of shortcuts. You can specify an initial position and size when you add a child view to the `AbsoluteLayout`:

```
absoluteLayout.Children.Add(child, new Rectangle(x, y, width, height));
```

Or, if you want the view automatically sized:

```
absoluteLayout.Children.Add(child, new Point(x, y));
```

However, if you need to change the position and size after the child has been added to the `Children` collection, you need to use the static `AbsoluteLayout.SetLayoutBounds` method.

As you can surmise, using `AbsoluteLayout` is more difficult than using `StackLayout`. In general it's much easier to let Xamarin.Forms and the other `Layout` classes handle much of the complexity of layout for you. But for some special applications, `AbsoluteLayout` is ideal.

One such special application is **DotMatrixClock**, which displays the digits of the current time using a simulated 5 × 7 dot matrix display. Each dot is a `BoxView`, individually sized and positioned on the screen and colored either red or light gray depending on whether the dot is on or off. Conceivably, the dots of this clock could be organized in nested `StackLayout` elements or a `Grid` (covered in the next chapter) but each `BoxView` needs to be given a size anyway. The sheer quantity and regularity of these views suggests that the programmer knows better how to arrange them on the screen than a layout that needs to perform the location calculations in a more generalized manner.

The `numberPatterns` and `colonPatterns` fields define the dot matrix patterns for the 10 digits and a colon separator:

```
class DotMatrixClockPage : ContentPage
{
    // 5 x 7 dot matrix patterns for 0 through 9.
    static readonly int[, ,] numberPatterns = new int[10, 7, 5]
    {
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 1, 1}, { 1, 0, 1, 0, 1},
            { 1, 1, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 0, 0}, { 0, 1, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0},
```

```
            { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0},
            { 0, 0, 1, 0, 0}, { 0, 1, 0, 0, 0}, { 1, 1, 1, 1, 1}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 0, 1, 0},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 0, 1, 0}, { 0, 0, 1, 1, 0}, { 0, 1, 0, 1, 0}, { 1, 0, 0, 1, 0},
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 0, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0}, { 0, 0, 0, 0, 1},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 1, 0}, { 0, 1, 0, 0, 0}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0},
            { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 1},
            { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 1, 1, 0, 0}
        },
    };

    // Dot matrix pattern for a colon.
    static readonly int[,] colonPattern = new int[7, 2]
    {
            { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }
    };

    // Total dots horizontally and vertically.
    const int horzDots = 41;
    const int vertDots = 7;

    // BoxView colors for on and off.
    static readonly Color colorOn = Color.Red;
    static readonly Color colorOff = new Color(0.5, 0.5, 0.5, 0.25);

    AbsoluteLayout absoluteLayout;

    // Box views for 6 digits, 7 rows, 5 columns.
    BoxView[, ,] digitBoxViews = new BoxView[6, 7, 5];
```

```
    // Box views for 2 colons, 7 rows, 2 columns.
    BoxView[, ,] colonBoxViews = new BoxView[2, 7, 2];

    …
}
```

Fields are also defined for arrays of `BoxView` objects for the 6 digits of the time (hour, minute, and second), and 2 colon separators. The total number of dots horizontally (set as `horzDots`) includes 5 dots for each of the 6 digits, 4 dots between digits for the 2 colons, and one dot width between the digits otherwise.

The program's constructor (shown below) creates a total of 238 `BoxView` objects and adds them to an `AbsoluteLayout` but also saves them in the field arrays. In theory, the `BoxView` objects can be referenced later by indexing the `Children` collection of the `AbsoluteLayout`. However, in the `Children` collection they just appear as a linear list. Storing them also in arrays allows them to be more easily identified and referenced.

```
class DotMatrixClockPage : ContentPage
{
    …

    public DotMatrixClockPage()
    {
        // Everything goes in the AbsoluteLayout.
        absoluteLayout = new AbsoluteLayout
        {
            VerticalOptions = LayoutOptions.Center
        };

        // Loop through the 6 digits in the clock.
        for (int digit = 0; digit < 6; digit++)
            for (int row = 0; row < 7; row++)
                for (int col = 0; col < 5; col++)
                {
                    // Create the BoxView and add to layout.
                    BoxView boxView = new BoxView();
                    digitBoxViews[digit, row, col] = boxView;
                    absoluteLayout.Children.Add(boxView);
                }

        // Loop through the 2 colons in the clock.
        for (int colon = 0; colon < 2; colon++)
            for (int row = 0; row < 7; row++)
                for (int col = 0; col < 2; col++)
                {
                    // Create the BoxView and set the color.
                    BoxView boxView = new BoxView
                    {
                        Color = colonPattern[row, col] == 1 ?
                                    colorOn : colorOff
```

```
                    };
                    colonBoxViews[colon, row, col] = boxView;
                    absoluteLayout.Children.Add(boxView);
                }

        // Set the page content to the AbsoluteLayout.
        this.Padding = new Thickness(10, 0);
        this.Content = absoluteLayout;

        // Set two event handlers.
        this.SizeChanged += OnPageSizeChanged;
        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimer);

        // Initialize with a manual call to OnTimer.
        OnTimer();
    }

    …

}
```

The `BoxView` objects for the time digits are not colored at all in the constructor, but those for the two colons are given a `Color` property based on the `colonPattern` array. The constructor does not give any `BoxView` objects locations or sizes. The positioning and sizing must be based on the size and orientation of the page.

The `DotMatrixClockPage` constructor concludes by setting event handlers for the page's `SizeChanged` event and a one-second timer.

The `SizeChanged` handler is responsible for positioning and sizing all the `BoxView` objects based on the width and height of the screen. To be as large as possible within the full width of the display, each `BoxView` occupies 1/41st of the screen width minus padding. That value is then used to position the various `BoxView` objects horizontally and vertically, with a size that is 85% of that value. Notice the calls to the static `AbsoluteLayout.SetLayoutBounds` methods:

```
class DotMatrixClockPage : ContentPage
{

    …

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        double increment = (this.Width - this.Padding.HorizontalThickness) / horzDots;

        // Dot size (so 0.15 * increment for dot gap).
        double size = 0.85 * increment;

        // Begin the LayoutBounds settings.
        double x = 0;

        for (int digit = 0; digit < 6; digit++)
        {
```

```
            for (int col = 0; col < 5; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    AbsoluteLayout.SetLayoutBounds(
                        digitBoxViews[digit, row, col],
                            new Rectangle(x, y, size, size));
                    y += increment;
                }
                x += increment;
            }
            x += increment;

            if (digit == 1 || digit == 3)
            {
                int colon = digit / 2;

                for (int col = 0; col < 2; col++)
                {
                    double y = 0;

                    for (int row = 0; row < 7; row++)
                    {
                        AbsoluteLayout.SetLayoutBounds(
                            colonBoxViews[colon, row, col],
                                new Rectangle(x, y, size, size));
                        y += increment;
                    }
                    x += increment;
                }
                x += increment;
            }
        }
    }

    …

}
```

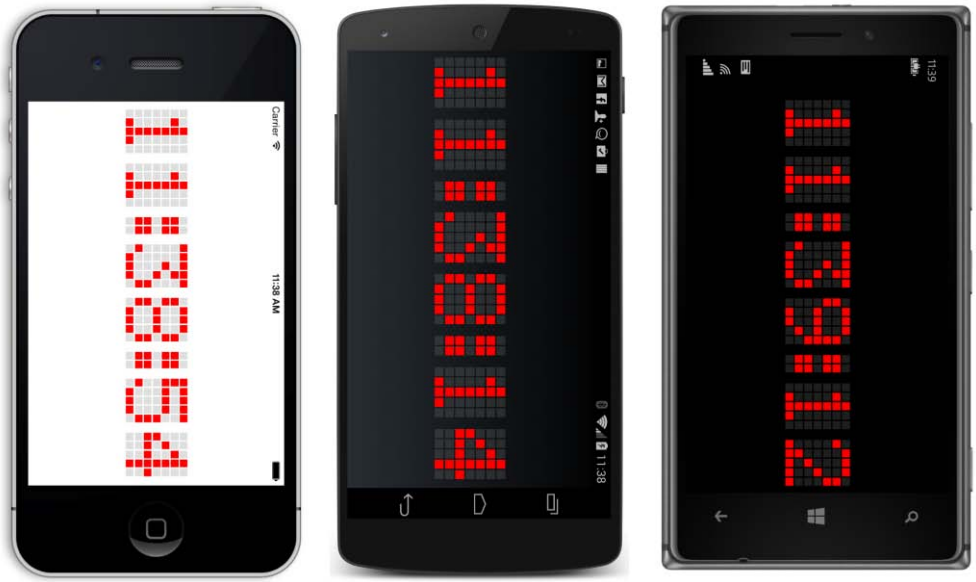
The `Device.StartTimer` event handler seems like it should be rather complex because it is responsible for setting the `Color` property of each `BoxView` based on the digits of the current time. However, the similarity between the definitions of the `numberPatterns` array and the `digitBoxViews` array makes it surprisingly straightforward:

```
class DotMatrixClockPage : ContentPage
{

    …

    bool OnTimer()
    {
        DateTime dateTime = DateTime.Now;
```

```
        // Convert 24-hour clock to 12-hour clock.
        int hour = (dateTime.Hour + 11) % 12 + 1;

        // Set the dot colors for each digit separately.
        SetDotMatrix(0, hour / 10);
        SetDotMatrix(1, hour % 10);
        SetDotMatrix(2, dateTime.Minute / 10);
        SetDotMatrix(3, dateTime.Minute % 10);
        SetDotMatrix(4, dateTime.Second / 10);
        SetDotMatrix(5, dateTime.Second % 10);
        return true;
    }

    void SetDotMatrix(int index, int digit)
    {
        for (int row = 0; row < 7; row++)
            for (int col = 0; col < 5; col++)
            {
                bool isOn = numberPatterns[digit, row, col] == 1;
                Color color = isOn ? colorOn : colorOff;
                digitBoxViews[index, row, col].Color = color;
            }
    }
}
```

And here's the result:

Of course, bigger is better, so you'll probably want to turn the phone (or the book) sideways for something large enough to read from across the room:



AbsoluteLayout proportionally

`AbsoluteLayout` is actually more sophisticated than the dot-matrix clock example might indicate. It includes a facility to automatically scale and position its children relative to its own size. You invoke this feature using members of the `AbsoluteLayoutFlags` enumeration:

- `None` (equal to 0)
- `XProportional` (1)
- `yProportional` (2)
- `PositionProportional` (3)
- `WidthProportional` (4)
- `HeightProportional` (8)
- `SizeProportional` (12)
- `All` (\xFFFFFFFF)

You can set a proportional flag on a child of `AbsoluteLayout` by using an attached bindable property applied through the following static method:

```
AbsoluteLayout.SetLayoutFlags(child, flags);
```

Or, you can use a third argument to the `Add` method that lets you specify the `Rectangle` bounds:

```
absoluteLayout.Children.Add(child, rect, flags);
```

For example, if you use the `SizeProportional` flag and set the width of the child to 0.25 and the height to 0.10, the child will be ¼ of the width of the `AbsoluteLayout` and 1/10th the height.

The `PositionProportional` flag is similar but it takes the size of the child into account: A position of (0, 0) puts the child in the upper-left corner, a position of (1, 1) puts the child in the lower-right corner, and a position of (0.5, 0.5) centers the child within the `AbsoluteLayout`.

Let's rewrite the **DotMatrixClock** program to use this feature and call it **ProportionalDotMatrix-Clock**. Much of the program remains the same. The `AbsoluteLayout` object still needs to be saved as a field, but the array of `BoxView` objects for the colons can be eliminated. All the positioning and sizing is now performed in the constructor based on an `AbsoluteLayout` that is assumed to have an aspect ratio of 41-to-7, which encompasses the 41 `BoxView` widths and 7 `BoxView` heights.

```
class ProportionalDotMatrixClockPage : ContentPage
{

    …

    AbsoluteLayout absoluteLayout;

    // Box views for 6 digits, 7 rows, 5 columns.
    BoxView[,,] digitBoxViews = new BoxView[6, 7, 5];

    public ProportionalDotMatrixClockPage()
    {
        // Everything goes in the AbsoluteLayout.
        absoluteLayout = new AbsoluteLayout
        {
            VerticalOptions = LayoutOptions.Center
        };

        // BoxView dot dimensions.
        const double height = 0.85 / vertDots;
        const double width = 0.85 / horzDots;

        // Create and assemble the BoxViews.
        double xIncrement = 1.0 / (horzDots - 1);
        double yIncrement = 1.0 / (vertDots - 1);
        double x = 0;

        for (int digit = 0; digit < 6; digit++)
        {
            for (int col = 0; col < 5; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the digit BoxView and add to layout.
                    BoxView boxView = new BoxView();
                    digitBoxViews[digit, row, col] = boxView;
```

```
                absoluteLayout.Children.Add(boxView,
                                            new Rectangle(x, y, width, height),
                                            AbsoluteLayoutFlags.All);
                y += yIncrement;
            }
            x += xIncrement;
        }
        x += xIncrement;

        // Colons between the hours, minutes, and seconds.
        if (digit == 1 || digit == 3)
        {
            int colon = digit / 2;

            for (int col = 0; col < 2; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the BoxView and set the color.
                    absoluteLayout.Children.Add(
                        new BoxView
                        {
                            Color = colonPattern[row, col] == 1 ?
                                        colorOn : colorOff
                        },
                        new Rectangle(x, y, width, height),
                        AbsoluteLayoutFlags.All);

                    y += yIncrement;
                }
                x += xIncrement;
            }
            x += xIncrement;
        }
    }

    // Set the page content to the AbsoluteLayout.
    this.Padding = new Thickness(10, 0);
    this.Content = absoluteLayout;

    // Set two event handlers.
    this.SizeChanged += OnPageSizeChanged;
    Device.StartTimer(TimeSpan.FromSeconds(1), OnTimer);

    // Initialize with a manual call to OnTimer.
    OnTimer();
}

void OnPageSizeChanged(object sender, EventArgs args)
{
    // No chance a display will have an aspect ratio > 41:7
    absoluteLayout.HeightRequest = vertDots * this.Width / horzDots;
```

```
    }

    …
}
```

Now the `SizeChanged` handler become just a single statement. The `AbsoluteLayout` is automatically stretched horizontally to fill the width of the page (minus the padding), so the `HeightRequest` really just sets the aspect ratio.

To position each `BoxView`, the constructor calculates proportional `xIncrement` and `yIncrement` values like so:

```
double xIncrement = 1.0 / (horzDots - 1);
double yIncrement = 1.0 / (vertDots - 1);
```

If the `AbsoluteLayout` worked a little differently—if it did not take the size of the child into account when using proportional positioning—then these denominators would be `horzDots` (set to 41) and `vertDots` (set to 7), and the 41 `BoxView` horizontal positions would range from 0 / 41 to 40 / 41. However, because `AbsoluteLayout` takes size into account, the rightmost `BoxView` must have a horizontal position of 1, which means that the horizontal positions must range from 0 / 40 to 40 / 40.

The size of each `BoxView`, however, is based on the width of 41 and height of 7:

```
const double height = 0.85 / vertDots;
const double width = 0.85 / horzDots;
```

The program is functionally the same as the one that did not use proportional positioning and sizing, but aside from height of the `AbsoluteLayout` itself, all the recalculations are automated.

The program has simpler event handling and fewer "moving parts." In that respect, this feature of `AbsoluteLayout` is similar to data-binding, which is explored in more depth in the next chapter.

CHAPTER 6

The interactive interface

Many Xamarin.Forms views are interactive and respond to touch gestures—tapping and dragging—and a few read input from the keyboard.

These interactive views incorporate paradigms and even names that are familiar to both users and programmers: Users can trigger commands with `Button`, specify a number from a range of values with `Slider` and `Stepper`, enter text from the phone's keyboard using `Entry` and `Editor`, and select items from a collection with `Picker`, `ListView` , and `TableView`.

This chapter is devoted to demonstrating the use of several of these interactive views. You've already seen the `Entry` and `Editor`, and detailed coverage of the various collection views won't appear until a chapter not in the Preview Edition, but this chapter explores several popular interactive views. In the process, the chapter goes deeper into data binding and introduces the powerful `Grid` layout.

View overview

The initial releases of Xamarin.Forms includes 19 instantiable classes that derive from `View` but not from `Layout`. You've already seen seven of these classes in the previous chapters: `Label`, `BoxView`, `Button`, `Entry`, `Editor`, `ListView`, and `Image`.

Seven views (including two you've already seen) allow the user to select or interact with basic .NET and C# data types:

Data Type	Views
`Double` / `double`	`Slider`, `Stepper`
`String` / `string`	`Entry`, `Editor`
`Boolean` / `bool`	`Switch`
`DateTime`	`DatePicker`, `TimePicker`

Much of the focus of this chapter is devoted to these views. The remaining views not discussed yet are:

- `SearchBar`, a single-line text entry with a button specifically for the purpose of initiating a search.
- `WebView`, to display web pages or HTML.

- `Picker`, to display selectable strings for program options.
- `TableView`, a list of items separated into categories that is flexible enough to be used for data, forms, menus, or settings.
- `ActivityIndicator` and `ProgressBar`, to indicate that a program is busy doing work.
- `OpenGLView`, which allows a program to display 2D and 3D graphics using the Open Graphics Library.

Slider, Stepper, and Switch

Both `Slider` and `Stepper` let the user select a numeric value from a range. They have nearly identical programming interfaces, but are very different visually and interactively. The **SliderStepperSwitch** program demonstrates these two views along with the Boolean `Switch` view. The program uses three instances of `Label` to display the currently selected values.

The `Slider` defines three public properties of type `double` named `Minimum`, `Maximum`, and `Value`. Visually, `Slider` is a horizontal bar with a slidable *thumb* that the user drags to change the `Value` property between the minimum and maximum. (The Xamarin.Forms `Slider` does not support a vertical orientation.) Whenever the `Value` property changes, the `Slider` fires a `ValueChanged` event indicating the new value.

The `SliderStepperSwitchPage` constructor begins by instantiating a `Slider` and attaching an event handler named `OnSliderValueChanged` to the `ValueChanged` event. The constructor continues by creating a `Label` that it saves as a field that can be referenced by the event handler:

```
class SliderStepperSwitchPage : ContentPage
{
    Label sliderValueLabel, stepperValueLabel, switchToggledLabel;

    public SliderStepperSwitchPage()
    {
        // Create a Slider and set event handler.
        Slider slider = new Slider
        {
            VerticalOptions = LayoutOptions.EndAndExpand
        };
        slider.ValueChanged += OnSliderValueChanged;

        // Create a Label to display the Slider value.
        sliderValueLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

            …
```

```
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        sliderValueLabel.Text = args.NewValue.ToString();
    }

    …

}
```

The first argument to the event handler is the object firing the event, in this case the `Slider`, and the second provides more information about this event. `ValueChangedEventArgs` has two properties of type double named `OldValue` and `NewValue`. The handler simply converts `NewValue` to a string and sets it to the `Text` property of the `Label`.

The `Stepper` has the same properties as `Slider` but adds an `Increment` property, also of type `double`. Just for variety, the `ValueChanged` handler for the `Stepper` is a local lambda function:

```
class SliderStepperSwitchPage : ContentPage
{
    Label sliderValueLabel, stepperValueLabel, switchToggledLabel;

    public SliderStepperSwitchPage()
    {

        …

        // Create a Stepper and set event handler.
        Stepper stepper = new Stepper
        {
            VerticalOptions = LayoutOptions.EndAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };
        stepper.ValueChanged +=
            (object sender, ValueChangedEventArgs args) =>
            {
                stepperValueLabel.Text = args.NewValue.ToString();
            };

        // Create a Label to display the Stepper value.
        stepperValueLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        …

    }
```

```
    …
}
```

Because this event handler is local to the constructor, it's not necessary for the `stepperValueLabel` to be saved as a field, but if it's not, you'll need to move the definition of the lambda function to be after the `Label` definition.

The `Switch` view has two states: on and off (or yes and no, or true and false, or however you want to think of them). `Switch` indicates the current state with a property named `IsToggled` of type `bool`, and it fires the `Toggled` event to indicate a change in this property. You might be inclined to give a `Switch` view a name of `switch`, but that's a C# keyword, so you'll want to pick something else:

```
class SliderStepperSwitchPage : ContentPage
{
    Label sliderValueLabel, stepperValueLabel, switchToggledLabel;

    public SliderStepperSwitchPage()
    {

        …

        // Create a Switch and set event handler.
        Switch switcher = new Switch
        {
            VerticalOptions = LayoutOptions.EndAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };
        switcher.Toggled += OnSwitcherToggled;

        // Create a Label to display the Switch value.
        switchToggledLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        …

    }

    …

    void OnSwitcherToggled(object sender, ToggledEventArgs args)
    {
        Switch switcher = (Switch)sender;
        switchToggledLabel.Text = switcher.IsToggled.ToString();
    }
}
```

Although `ToggledEventArgs` has a `Value` property of type `bool` that indicates the new state of the `Switch`, the event handler chooses to cast the first argument to the `Switch` object and obtain the

value from the `IsToggled` property. That's valid as well.

The constructor concludes by assembling all six views in a `StackLayout`:

```
public SliderStepperSwitchPage()
{

    …

    // Put them all in a StackLayout.
    this.Content = new StackLayout
    {
        Children =
        {
            slider,
            sliderValueLabel,
            stepper,
            stepperValueLabel,
            switcher,
            switchToggledLabel,
        }
    };
}
```

The `VerticalOptions` setting of each view has been set to visually group each interactive view with its corresponding `Label`. But when the program first starts up, the `Text` property of each `Label` is `null`, and hence the `Label` has zero height. As soon as you begin manipulating the views, the `Text` properties are set from the event handlers, and the views will shift vertical position slightly. As you can see, these three views have very different appearances on the three platforms, but they are functionally pretty much the same:

A little experimentation reveals that the default `Minimum` and `Maximum` values are 0 and 1 for the `Slider`, but 0 and 100 for the `Stepper`, and of course you can change them. The `Stepper` has a default Increment value of 1.

If you don't like the many decimal places displayed for the `Slider` on the iPhone and Windows Phone, you can easily fix that with a formatting string appropriate for `double`:

```
sliderValueLabel.Text = args.NewValue.ToString("F2");
```

Suppose you want to initialize the `Value` property of the `Slider` after instantiating it:

```
Slider slider = new Slider
{
    VerticalOptions = LayoutOptions.EndAndExpand
};
slider.ValueChanged += OnSliderValueChanged;
slider.Value = 0.5;
```

This code will cause a `NullReferenceException` to be raised. The problem is that setting the `Value` property triggers a call to the `OnSliderValueChanged` handler, which references a `Label` that hasn't been created yet. You can avoid this problem by setting the `Value` property before attaching the event handler, or after instantiating the `Label`. Or, the event handler can check if the `Label` object is not `null` before attempting to set its `Text` property.

Instead of setting the `Value` property of `Slider` to 0.5, try this:

```
slider.Value = 0;
```

This statement will *not* result in a `NullReferenceException` because it will not trigger the `ValueChanged` event to be fired. The event is called `ValueChanged` and not `ValueSet` for a reason: The event is not fired unless the `Value` property has actually changed.

If you want the `Label` views to display the initial values at program startup, your code can set the `Text` properties to the appropriate values when the `Label` views are created. Or, you can "fake" an event by calling the event handlers directly, perhaps at the end of the constructor after everything has been created.

Or, you can use data bindings.

Data binding

The **SliderStepperSwitch** program sets event handlers so that the `Text` property of a `Label` tracks the value of the interactive view. Tasks such as this can be automated with the same data-binding techniques you saw in the **NoteTaker** programs in Chapter 3 and 4.

As you learned at that time, data bindings link the value of a property of one object with the value of a property of another object to keep the values synchronized. These two objects are distinguished as

a *source* and a *target*. In the simplest case, the source is an object with a property that changes value, and the target is another object with a property that is changed as a result. However, the relationship between source and target can be reversed: Changes in the target can update the source, or the source and target can update each other.

A data-binding target object must derive from `BindableObject` and the target property must be backed by a bindable property. This restriction is imposed by the data-binding API. A data-binding source should implement the `INotifyPropertyChanged` interface. This is the standard way for a data-binding source to indicate when its property has changed.

In real-world programming, probably the most common forms of data bindings are similar to those illustrated by the **NoteTaker** programs: The data-binding target is a view, such as the `Entry` and `Editor` views. All user-interface classes in Xamarin.Forms derive from `BindableObject` and many of their properties are backed by bindable properties. Typically, the data-binding source is a data object that implements `INotifyPropertyChanged`.

However, `BindableObject` also implements `INotifyPropertyChanged`, which means that Xamarin.Forms views can be data-binding sources as well as targets. Data bindings between views on the same page are sometimes called *view-to-view bindings*.

Suppose a view named `sourceView` has a property named `SourceProp`. Suppose also that `targetView` is an instance of the `GreatView` class that defines a property named `TargetProp` backed by a `BindableProperty` named `GreatView.TargetPropProperty`. You can define a data binding to keep the `SourceProp` property updated with the value of the `TargetProp` property with two statements. The first statement associates the two objects by setting the `BindingContext` property of the target object to the source object:

```
targetView.BindingContext = sourceView;
```

This `BindingContext` property is defined by `BindableObject`. The second statement involves the `SetBinding` method (also defined by `BindableObject`), which links the two properties:

```
targetView.SetBinding(GreatView.TargetPropProperty, "SourceProp");
```

`BindableObject` defines one `SetBinding` method, and that's supplemented by two additional `SetBinding` extension methods defined in the `BindableObjectExtensions` class. The SetBinding call shown in the example above is defined in that extensions class. You'll see more complex alternatives later in this chapter.

You can pretty much guess what's going on behind the scenes: A handler for the `PropertyChanged` event is attached to `sourceView`, and that handler looks for a property named "SourceProp". When that property changes, the event handler can use .NET reflection to obtain the value of `sourceView.SourceProp` and then call `SetValue` on the `targetView` object with an argument of `GreatView.TargetPropProperty`.

The **SliderStepperSwitchBindings** program features a page constructor that begins by assembling the now familiar `Slider`, `Stepper`, `Switch`, and `Label` views but does not save any objects as fields

or install any event handlers:

```
class SliderStepperSwitchBindingsPage : ContentPage
{
    public SliderStepperSwitchBindingsPage()
    {
        // Create a Slider.
        Slider slider = new Slider
        {
            VerticalOptions = LayoutOptions.EndAndExpand
        };

        // Create a Label to display the Slider value.
        Label sliderValueLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        // Create a Stepper.
        Stepper stepper = new Stepper
        {
            VerticalOptions = LayoutOptions.EndAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        // Create a Label to display the Stepper value.
        Label stepperValueLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center,
        };

        // Create a Switch.
        Switch switcher = new Switch
        {
            IsToggled = true,
            VerticalOptions = LayoutOptions.EndAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        // Create a Label to display the Switch value.
        Label switchToggledLabel = new Label
        {
            Font = Font.SystemFontOfSize(NamedSize.Large),
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        // Put them all in a StackLayout.
        this.Content = new StackLayout
        {
```

```
            Children =
            {
                slider,
                sliderValueLabel,
                stepper,
                stepperValueLabel,
                switcher,
                switchToggledLabel,
            }
        };

        …

    }
}
```

Notice the ellipsis at the bottom. That's where the data bindings are defined, and you'll see that they go beyond simply displaying the values of the views.

Let's use a data binding to control the opacity of a `Label`. The `VisualElement` class defines a property named `Opacity` of type `double` that ranges from 0 to 1, and the `Value` property of the `Slider` is also of type `double`, and also ranges from 0 to 1. Here's a data binding that targets the `Opacity` property of the first `Label` from the `Value` property of the `Slider`:

```
sliderValueLabel.BindingContext = slider;
sliderValueLabel.SetBinding (Label.OpacityProperty, "Value");
```

With this data binding in place, the `Opacity` of the first `Label` will vary from fully transparent to fully opaque as the `Slider` value changes.

Let's try another: The `VisualElement` class defines an `IsEnabled` property of type `bool`, and the `IsToggled` property of the `Switch` is also of type `bool`. These two data bindings target the `IsEnabled` properties of the `Slider` and `Stepper` from the source `IsToggled` property of the `Switch`:

```
slider.BindingContext = switcher;
slider.SetBinding (Slider.IsEnabledProperty, "IsToggled");

stepper.BindingContext = switcher;
stepper.SetBinding (Stepper.IsEnabledProperty, "IsToggled");
```

When the `Switch` is toggled off, the `Slider` and `Stepper` are dimmed out a bit and do not respond to user input. To avoid confusion by presenting non-functional `Slider` and `Stepper` views at program startup, the `IsToggled` property of the `Switch` has been initialized to `true`.

What happens when the data types of the source and target properties are not the same? Automatic data conversions are *not* performed, but you can supply a small class that converts between the two data types. To reference a data-converter class, it's easiest to use a different `SetBinding` call and making use of the `Binding` class:

```
Binding binding = new Binding ("SourceProp", …);
```

```
targetView.SetBinding (GreatView.TargetPropProperty, binding);
```

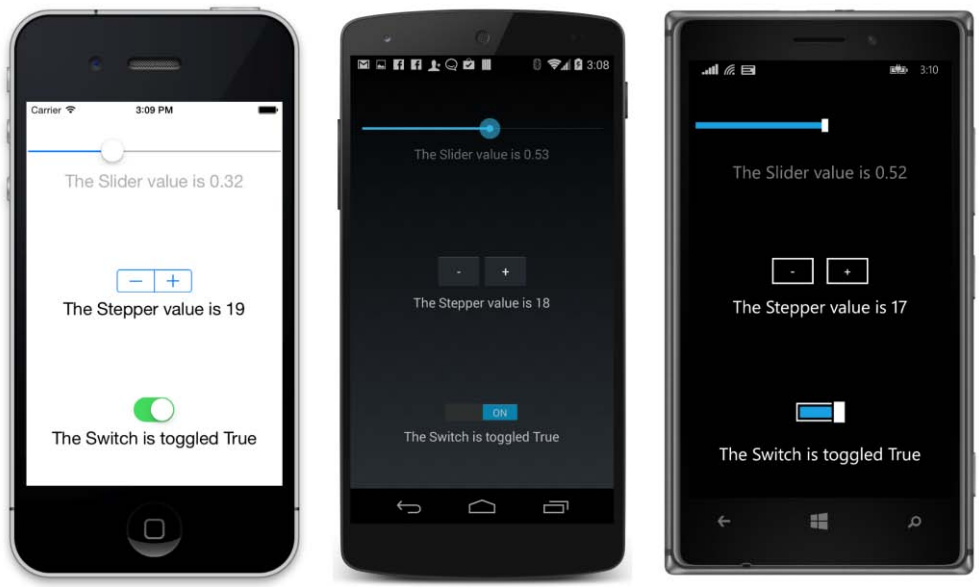
You'll see examples at the end of this chapter.

A special facility exists for data conversion when the target property is of type `string`, which is the case for the labels. You can direct the binding logic to use `String.Format` for this conversion by simply supplying a standard .NET formatting string. Here are the three bindings for the labels. The first one doesn't require setting the `BindingContext` because it's already set:

```
sliderValueLabel.SetBinding (Label.TextProperty,
    new Binding ("Value", BindingMode.Default, null, null,
                 "The Slider value is {0:F2}"));

stepperValueLabel.BindingContext = stepper;
stepperValueLabel.SetBinding (Label.TextProperty,
    new Binding ("Value", BindingMode.Default, null, null,
                 "The Stepper value is {0}"));

switchToggledLabel.BindingContext = switcher;
switchToggledLabel.SetBinding (Label.TextProperty,
    new Binding ("IsToggled", BindingMode.Default, null, null,
                 "The Switch is toggled {0}"));
```

When you run this program you'll notice a big change in the labels compared with the earlier programs: All three `Label` views display valid text before the views are manipulated (except that you won't be able to see the one with the `Opacity` of zero). When a data binding is first defined, it accesses the value of the source property and sets the value to the target.

The three screenshots show this enhanced text and the translucency of the first label:



Strictly speaking, data bindings are not a necessary part of the Xamarin.Forms interface. Everything you can do with a data binding you can do with explicit event handlers. But data bindings can install and manage these event handlers for you, so they allow you to write code with less event processing, and which consequently appears to have fewer "moving parts."

One element of a data binding definition that is vulnerable to bugs is the source property name, which you indicate with a text string. Visual Studio and Xamarin Studio won't warn you if you've spelled it wrong, but the binding won't work, and the problem might be difficult to find.

You can avoid spelling errors with a generic form of the `SetBinding` method. Earlier you saw a binding to the `Opacity` property of the `Label` from the `Value` property of the `Slider` defined like this:

```
sliderValueLabel.SetBinding (Label.OpacityProperty, "Value");
```

You can alternately use this form:

```
sliderValueLabel.SetBinding<Slider> (Label.OpacityProperty, src => src.Value);
```

The generic argument is `Slider`, which is the type of the source object, and the source `Value` property is indicated as a lambda expression that accesses the property. The method uses .NET reflection to obtain the string property name.

You can do something similar with the longer version of the `SetBinding` method. Here's the statement that binds the `Text` property of the `Label` to the `Value` property of the `Slider`:

```
sliderValueLabel.SetBinding (Label.TextProperty,
    new Binding ("Value", BindingMode.Default, null, null,
                 "The Slider value is {0:F2}"));
```

Rather than using a `Binding` constructor as the second argument you can use a static (and generic) `Binding.Create` method:

```
sliderValueLabel.SetBinding (Label.TextProperty,
    Binding.Create<Slider> (src => src.Value, BindingMode.Default, null, null,
                            "The Slider value is {0:F2}"));
```

These alternative forms are much preferred if you're also developing the code that defines the source properties, because you can rename these source properties without fear of breaking the code defining the data bindings.

Here's a new version of the **WhatSize** program that uses both forms of `Binding` creation (a practice that in real life might make people who need to work on the program sometime later wonder at this apparent intention of inconsistency):

```
class WhatSizeBindingsPage : ContentPage
{
    public WhatSizeBindingsPage()
    {
        Font font = Font.SystemFontOfSize(NamedSize.Large);
```

```
        // Create Label views for displaying page width and height.
        Label widthLabel = new Label
        {
            Font = font
        };

        Label heightLabel = new Label
        {
            Font = font
        };

        // Define bindings on Labels.
        widthLabel.BindingContext = this;
        widthLabel.SetBinding (Label.TextProperty,
            Binding.Create<ContentPage> (src => src.Width,
                                         BindingMode.OneWay,
                                         null, null, "{0:F0}"));

        heightLabel.BindingContext = this;
        heightLabel.SetBinding (Label.TextProperty,
            new Binding ("Height",
                         BindingMode.OneWay,
                         null, null, "{0:F0}"));

        // Assemble a horizontal StackLayout with the three labels.
        this.Content = new StackLayout
        {
            Orientation = StackOrientation.Horizontal,
            Spacing = 0,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Children =
            {
                widthLabel,
                new Label
                {
                    Text = " x ",
                    Font = font
                },
                heightLabel
            }
        };
    }
}
```

Notice the use of a horizontal `StackLayout` with no spacing to visually concatenate three pieces of text so they appear to be a single line.

As you shift the phone between portrait and landscape, the value updates entirely through the data bindings.

Date and time selection

While the information from the `DateTime.Now` property can be displayed in a variety of ways, sometimes applications need to allow the user to select a date or time. For this purpose, Xamarin.Forms includes `DatePicker` and `TimePicker` views.

These are very similar. The two views display a date or time, and tapping the view invokes the platform-specific date or time selector. On the iPhone and Android, these selectors take over part of the screen; on Windows Phone, it takes over the whole screen. The user then dials in a new date or time and indicates completion.

`DatePicker` has three properties of type `DateTime`:

- `MinimumDate`, initialized to January 1, 1900
- `MaximumDate`, initialized to December 31, 2100
- `Date`, initialized to `DateTime.Today`

A program can set these properties to whatever it wants. The `Date` property also reflects the user's selection.

The `DatePicker` displays the selected date using the normal `ToString` method, but you can set the `Format` property of the view to a custom .NET formatting string. The initial value is "d"—the "short date" format.

Here's a program that lets you select two dates and calculates the number of days between them:

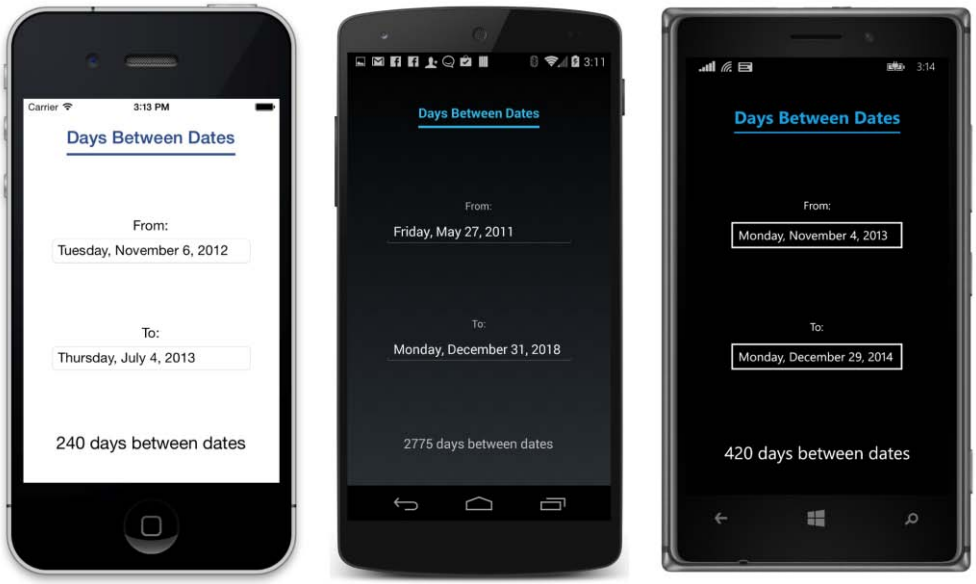
```
class DaysBetweenDatesPage : ContentPage
{
    static readonly string dateFormat = "D";
    DatePicker fromDatePicker, toDatePicker;
    Label resultLabel;

    public DaysBetweenDatesPage()
    {
        // Create DatePicker views.
        fromDatePicker = new DatePicker
        {
            Format = dateFormat,
            HorizontalOptions = LayoutOptions.Center
        };
        fromDatePicker.DateSelected += OnDateSelected;

        toDatePicker = new DatePicker
        {
            Format = dateFormat,
            HorizontalOptions = LayoutOptions.Center
        };
        toDatePicker.DateSelected += OnDateSelected;
```

```
// Create Label for displaying result.
resultLabel = new Label
{
    Font = Font.SystemFontOfSize(NamedSize.Large),
    HorizontalOptions = LayoutOptions.Center,
    VerticalOptions = LayoutOptions.CenterAndExpand
};

// Build the page.
this.Content = new StackLayout
{
    Children =
    {
        // Program title and underline.
        new StackLayout
        {
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center,
            Children =
            {
                new Label
                {
                    Text = "Days Between Dates",
                    Font = Font.SystemFontOfSize(NamedSize.Large, FontAttributes.Bold),
                    TextColor = Color.Accent
                },
                new BoxView
                {
                    HeightRequest = 3,
                    Color = Color.Accent
                }
            }
        },

        // "From" DatePicker view.
        new StackLayout
        {
            VerticalOptions = LayoutOptions.CenterAndExpand,
            Children =
            {
                new Label
                {
                    Text = "From:",
                    HorizontalOptions = LayoutOptions.Center
                },

                fromDatePicker
            }
        },

        // "To" DatePicker view.
        new StackLayout
        {
```

```
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    Children =
                    {
                        new Label
                        {
                            Text = "To:",
                            HorizontalOptions = LayoutOptions.Center
                        },

                        toDatePicker
                    }
                },

                // Label for result.
                resultLabel
            }
        };

        // Initialize results display.
        OnDateSelected(null, null);
    }

    void OnDateSelected(object sender, DateChangedEventArgs args)
    {
        int days = (int)Math.Round((toDatePicker.Date - fromDatePicker.Date).TotalDays);
        resultLabel.Text = String.Format("{0:F0} day{1} between dates",
                                         days, days == 1 ? "" : "s");
    }
}
```

The Windows Phone implementation of `TimePicker` uses some toolbar icons in the *Toolkit.Content* folder. If the icons do not display correctly, go into that folder and give the PNG files you'll find there a **Build Action** of **Content**.

Using a `StackLayout` within a `StackLayout` in combination with the `CenterAndExpand` layout spaces the components nicely in the vertical space of the page:



The `TimePicker` is somewhat simpler than `DatePicker`. It only defines `Time` and `Format` properties, and doesn't include an event to indicate a new `Time` value. If you need to be notified, you can install a handler for the `PropertyChanged` event.

Custom views: A radio button

You can create custom views in Xamarin.Forms. A later chapter (not in the Preview edition) will show you how to create custom views by writing special classes, called *renderers*, specific to each platform.

However, it's also possible to create custom views right in Xamarin.Forms. One approach is to derive a class from `ContentView`, and set its `Content` property to a `StackLayout` (for example) and then assemble one or more views on that. You'll probably also need to define a property or two, and possibly some events, but you'll want to take advantage of the existing infrastructure established by the `BindableObject` and `BindableProperty` classes.

Let's try to create a classic *radio button*. The radio button is named after radio station-selector buttons in old cars. These radios feature a row of pushbuttons, but only one button can be depressed at a time, so pressing a button makes the previously depressed button pop up, giving great joy to young kids who play with the radio.

Functionally, a radio button is similar to the Xamarin.Forms `Switch` in that it toggles between on and off states. However, radio buttons always appear in a group. The radio buttons are mutually exclusive, so selecting any radio button automatically deselects all the other buttons.

This custom `RadioButton` view shown below defines three properties, although more could be

added at a later time: `Text` (similar to `Button`), `Font` (similar to `Button`), and `IsToggled` (similar to `Switch`). All three of these properties are backed by bindable properties. Also similar to `Switch` is a `Toggled` event that is fired whenever the `IsToggled` property changes.

Multiple `RadioButton` views are usually assembled in a `StackLayout` or another layout. Therefore, any `RadioButton` that is toggled on can obtain its parent view, and if that view derives from `Layout<T>` and has a `Children` property, it can access its siblings, and toggle those off. (However, this common-parent assumption won't be valid if you use a separate `ContentView` or `Frame` as a parent to each `RadioButton`. In that case the `RadioButton` objects will not have a common parent, and you might instead be forced to implement another property, perhaps named `RadioGroup`, and use that instead.)

Visually, this `RadioButton` consists of two side-by-side `Label` views. The `Label` view on the right displays the `Text` property. The `Label` view on the left displays either Unicode \u25CB (○) or \u25C9 (◉), depending on the `IsToggled` value.

Here are the private fields and the portion of the constructor that assembles the visuals of the `RadioButton`:

```
class RadioButton : ContentView
{
    static readonly string checkOff = "¥x25CB";
    static readonly string checkOn = "¥u25C9";
    Label checkLabel;

    public RadioButton()
    {
        // Assemble the view.
        checkLabel = new Label
        {
            Text = checkOff
        };

        Label textLabel = new Label();

        this.Content = new StackLayout
        {
            Orientation = StackOrientation.Horizontal,
            Children =
            {
                checkLabel,
                textLabel
            }
        };

        …

    }

    …
```

```
}
```

These two `Label` views have `Text` properties, but the `ContentView` from which `RadioButton` derives does not have a `Text` property, so `RadioButton` must define one. Moreover, to allow data binding and other Xamarin.Forms features, `RadioButton` must also define a `BindableProperty` object named `TextProperty`.

This `TextProperty` is a field defined as `public` and `static`, and it should also be defined as `readonly`, which means that it must be set in the field definition or a static constructor. The `BindableProperty` class has static `Create` methods that create `BindableProperty` objects:

```
class RadioButton : ContentView
{
    …
    public static readonly BindableProperty TextProperty =
        BindableProperty.Create("Text",                 // property name
                                typeof(string),         // property type
                                typeof(RadioButton),    // this type
                                null);                  // initial value
    …
}
```

This is a relatively simple definition of a bindable property. The property name and type are supplied, as well as the type of the class defining the property and an initial value. More extensive use of the `BindableProperty.Create` method allows you to specify methods that are called when the property is about to change, and when it has changed, and for validation and coercion.

The normal `Text` property then references the bindable property in a very standard way:

```
class RadioButton : ContentView
{
    …
    public string Text
    {
        set { SetValue(TextProperty, value); }
        get { return (string)GetValue(TextProperty); }
    }
    …
}
```

You'll notice that the `BindableProperty.Create` method requires the property name as a text string, and if you misspell it, the property won't work right and it will be difficult to track down the problem. For that reason, generic alternatives of the `Create` method are available that derive the property name and type for you. The `RadioButton` uses one of these methods to define the `Font` property:

```
class RadioButton : ContentView
{
    …
    // Define the Font bindable property and property.
    public static readonly BindableProperty FontProperty =
```

```
        BindableProperty.Create<RadioButton, Font>
            (radio => radio.Font,                          // property
                Font.SystemFontOfSize(NamedSize.Large));   // initial value

    public Font Font
    {
        set { SetValue(FontProperty, value); }
        get { return (Font)GetValue(FontProperty); }
    }
    …
}
```

The `Create` method uses reflection to obtain the string rendition of the Font property name. The `Font` property is initialized to the `Large` size because the `Medium` size doesn't provide an adequate touch target.

Now the `RadioButton` has `Text` and `Font` properties but the values of these properties must be set on the two `Label` views the `RadioButton` has created in its constructor. This could happen in property-changed handlers for the `Text` and `Font` properties just defined, but data bindings provide a simpler approach. In the following data bindings defined in the `RadioButton` constructor, the targets are the `Label` views, and the source is the `RadioButton` itself:

```
class RadioButton : ContentView
{
    …
    public RadioButton()
    {
        …
        // Set the bindings.
        checkLabel.BindingContext = this;
        checkLabel.SetBinding(Label.FontProperty, "Font");

        textLabel.BindingContext = this;
        textLabel.SetBinding(Label.TextProperty, "Text");
        textLabel.SetBinding(Label.FontProperty, "Font");
        …
    }
    …
}
```

The `RadioButton` also requires a `Toggled` event and an `IsToggled` property backed by an `IsToggledProperty` bindable property. This definition of the `BindableProperty` object contains a reference to a method called when the property changes:

```
class RadioButton : ContentView
{
    …
    // Define the Toggled event, IsToggled bindable property and property.
    public event EventHandler<ToggledEventArgs> Toggled;

    public static readonly BindableProperty IsToggledProperty =
        BindableProperty.Create("IsToggled",
```

```
                                    typeof(bool),           // property type
                                    typeof(RadioButton),    // this type
                                    false,                  // initial value
                                    BindingMode.TwoWay,     // default binding
                                    null,                   // validation
                                    OnIsToggledPropertyChanged);

    public bool IsToggled
    {
        get { return (bool)GetValue(IsToggledProperty); }
        set { SetValue(IsToggledProperty, value); }
    }
    …
}
```

The `OnIsToggledPropertyChanged` method referenced in the static `BindableProperty.Create` method is called when the `IsToggled` property changes. However, because the method is referenced in a static field definition, the method itself must also be static. It has a signature like this:

```
    static void OnIsToggledPropertyChanged(BindableObject sender,
                                           object oldValue, object newValue)
    {
        …
    }
```

This method is called whenever the `IsToggled` property of a `RadioButton` instance changes. However, this method is static! If there are several instances of `RadioButton` (and that's normal), how can this method know which `RadioButton` has been toggled?

Simple: The first argument is the actual `RadioButton` instance whose `IsToggled` property has changed, and the second and third arguments indicate the old and new values. This means you can safely cast the first argument to an object of type `RadioButton`:

```
    static void OnIsToggledPropertyChanged(BindableObject sender,
                                           object oldValue, object newValue)
    {
        // Get the object whose property is changing.
        RadioButton radioButton = (RadioButton)sender;
        …
    }
```

And you can take it from there.

But it's often convenient for this static method to call an instance method, and for the instance method to do all the work. That's how it's done in `RadioButton`. There are basically three jobs this method must perform:

- Set the appropriate character to indicate a checked or unchecked state.
- Fire the `IsToggled` event.

- If `IsToggled` is `true`, set `IsToggled` to `false` on all `RadioButton` siblings.

Here's both the static and instance versions of the `IsToggled` property changed handler:

```
class RadioButton : ContentView
{
    …
    // Property changed handler for the IsToggled property.
    static void OnIsToggledPropertyChanged(BindableObject sender,
                                           object oldValue, object newValue)
    {
        // Get the object whose property is changing.
        RadioButton radioButton = (RadioButton)sender;
        radioButton.OnIsToggledPropertyChanged((bool)oldValue, (bool)newValue);
    }

    void OnIsToggledPropertyChanged(bool oldValue, bool newValue)
    {
        this.checkLabel.Text = newValue ? checkOn : checkOff;

        // Fire the event.
        if (this.Toggled != null)
        {
            this.Toggled(this, new ToggledEventArgs(this.IsToggled));
        }

        // If toggled on, toggle off all siblings.
        if (this.IsToggled)
        {
            Layout<View> parent = this.ParentView as Layout<View>;

            if (parent != null)
            {
                foreach (View view in parent.Children)
                {
                    if (view is RadioButton && view != this)
                    {
                        ((RadioButton)view).IsToggled = false;
                    }
                }
            }
        }
    }
}
```

There's only one piece missing: `RadioButton` doesn't yet respond to touch. Whenever a `RadioButton` is tapped, it must set its `IsToggled` property to `true`.

A Xamarin.Forms view can obtain touch input through a `GestureRecognizer` object. For taps, you create an object of type `TapGestureRecognizer` with a callback method that is called when the taps occur. (This is the only type of `GestureRecognizer` currently available.) Then add this `TapGestureRecognizer` to the `GestureRecognizers` collection of the view.

`RadioButton` defines `TappedCallback` as an anonymous lambda function:

```
class RadioButton : ContentView
{
    …
    public RadioButton()
    {
        …
        // Install a Tap recognizer.
        TapGestureRecognizer recognizer = new TapGestureRecognizer
        {
            Parent = this,
            NumberOfTapsRequired = 1,
            TappedCallback = (View view, Object args) =>
                {
                    ((RadioButton)view).IsToggled = true;
                }
        };

        this.GestureRecognizers.Add (recognizer);
    }
    …
}
```

In this example, the single statement of the `TappedCallback` method can even be a little simpler. Rather than cast the first argument of the method to a `RadioButton`, the callback could use `this`:

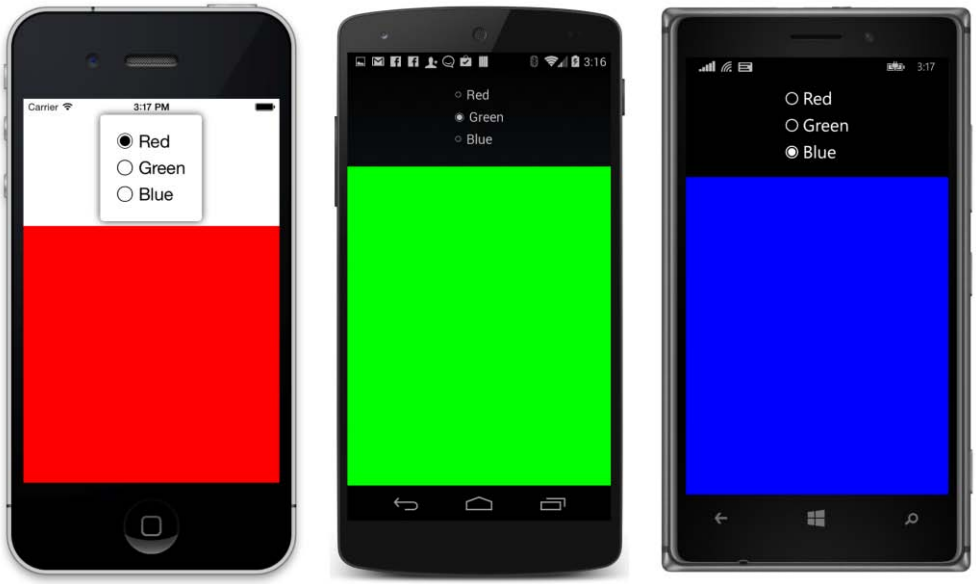
```
this.IsToggled = true;
```

Let's try it out. The **RadioColors** program includes the `RadioButton` class and creates three instances of `RadioButton` that share the same event handler. It uses these to set the color of a `BoxView`:

```
class RadioColorsPage : ContentPage
{
    BoxView boxView;

    public RadioColorsPage()
    {
        StackLayout radioStack = new StackLayout();

        // Three RadioButtons in the StackLayout.
        RadioButton radioButton = new RadioButton
        {
            Text = "Red",
            StyleId = "#FF0000"
        };
        radioButton.Toggled += OnRadioButtonToggled;
        radioStack.Children.Add(radioButton);

        radioButton = new RadioButton
        {
            Text = "Green",
            StyleId = "#00FF00"
```

```
        };
        radioButton.Toggled += OnRadioButtonToggled;
        radioStack.Children.Add(radioButton);

        radioButton = new RadioButton
        {
            Text = "Blue",
            StyleId = "#0000FF"
        };
        radioButton.Toggled += OnRadioButtonToggled;
        radioStack.Children.Add(radioButton);

        // A BoxView to display the selected color.
        boxView = new BoxView
        {
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        this.Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);

        // Build the page.
        this.Content = new StackLayout
        {
            Children =
            {
                new Frame
                {
                    HorizontalOptions = LayoutOptions.Center,
                    Content = radioStack
                },
                boxView
            }
        };

        // Initialize.
        ((RadioButton)radioStack.Children[0]).IsToggled = true;
    }

    void OnRadioButtonToggled(object sender, ToggledEventArgs args)
    {
        if (args.Value)
        {
            // Set the color from the StyleId string.
            boxView.Color = Color.FromHex(((RadioButton)sender).StyleId);
        }
    }
}
```

Notice that the `StyleId` property of each `RadioButton` is set to a hexadecimal string representation of the color. The `OnRadioButtonToggled` event handler simply passes this to the `Color.FromHex` method to set the new color of the `BoxView`. (This is a rather unorthodox use of the `StyleId`. If you want to store information such as a `Color` value with a view, it's perhaps better to derive from the view and add a simple read/write property of the desired type.)



Of course, you might have an entirely different idea about the visuals of a radio button. For example, you might want to use simple `Label` views and set the font to bold for a toggled-on state or the `Opacity` property to 0.5 for the toggled-off state. Helping you with this is the goal of the following extension methods:

```
static class RadioExtensions
{
    class Info
    {
        public bool toggledState;
        public Action<View> toggledHandler;
    }

    static Dictionary<View, Info> instances = new Dictionary<View, Info>();

    public static void AddRadioToggler(this View view, Action<View> toggledHandler)
    {
        // Add View to dictionary.
        instances.Add(view, new Info
        {
            toggledState = false,
            toggledHandler = toggledHandler
        });

        // Add a gesture recognizer for tap.
        view.GestureRecognizers.Add(
            new TapGestureRecognizer((View tappedView) =>
            {
                tappedView.SetRadioState(true);
            }));
    }
```

```
    public static void SetRadioState(this View view, bool isToggled)
    {
        // Check if the property is actually changing.
        if (instances[view].toggledState != isToggled)
        {
            // Set the new value.
            instances[view].toggledState = isToggled;

            // Fire the handler.
            instances[view].toggledHandler(view);

            // If being toggled, untoggle all the siblings.
            if (isToggled)
            {
                Layout<View> parent = view.ParentView as Layout<View>;

                if (parent != null)
                {
                    foreach (View sibling in parent.Children)
                    {
                        if (sibling != view && instances.ContainsKey(sibling))
                        {
                            sibling.SetRadioState(false);
                        }
                    }
                }
            }
        }
    }

    public static bool GetRadioState(this View view)
    {
        return instances[view].toggledState;
    }
}
```

You can call `AttachRadioToggler` on any `View` derivative and supply a change handler. The `SetRadioState` and `GetRadioState` extension methods are necessary because there is no such thing as an extension property.

In response to a call to the `toggledHandler` method, a program can change the `View` in whatever way it wants. For example, if you're using a `Label` for the items, you can change the font or the opacity. You'll see an example in the upcoming **ColorScroll** program.

Mastering the Grid

The `Grid` is a powerful layout mechanism that organizes its children into rows and columns of cells. At first the `Grid` seems to resemble the HTML `table`, but the `table` is for presentation purposes, while the `Grid` is solely for layout. There is no concept of a heading in a `Grid`, for example.

`Grid` redefines its `Children` property to include several custom methods for adding views to the `Children` collection. Here's one:

```
grid.Children.Add (child, col, row);
```

That puts the child view object in the specified zero-based column and row of the `Grid`. For example, the call

```
grid.Children.Add (child, 2, 3);
```

puts the child view in column 2 (which is actually the 3rd column) and row 3 (which is the 4th row). You do not need to predefine the number of columns and rows; the `Grid` automatically accommodates itself to these `Add` calls.

By default, when the `Grid` lays out its children, each column gets a width that is equal to the width of the widest child in that column. Each row gets a height that is equal to the height of the tallest child in that row. Columns and rows are separated based on the `ColumnSpacing` and `RowSpacing` properties, which have default values of 6.

The actual definition of this `Add` method uses the words `left` and `top` rather than `col` and `row`:

```
grid.Children.Add (child, left, top);
```

That terminology makes more sense when you see this `Add` method:

```
grid.Children.Add (child, left, right, top, bottom);
```

This form allows child views to span one or more columns, or one or more rows. When setting `right` and `bottom`, use the number of the *next* column or row that the child view does not occupy. In other words, the number of columns spanned by a view is equal to `right` minus `left`, which must be greater than or equal to 1.

For example:

```
grid.Children.Add (child, 2, 5, 7, 8);
```

The child view will occupy columns 2, 3, and 4 but only row 7.

We are now ready to rewrite the **SimplestKeypad** program from Chapter 2 to use a `Grid`. The new version is called **KeyboardGrid** and the constructor of the `KeypadGridPage` looks like this:

```
class KeypadGridPage : ContentPage
{
    Label displayLabel;
    Button backspaceButton;

    public KeypadGridPage()
    {
        // Create a centered grid for the entire keypad.
        Grid grid = new Grid
        {
            VerticalOptions = LayoutOptions.Center,
```

```
            HorizontalOptions = LayoutOptions.Center
        };

        // First row with Label and backspace Button.
        displayLabel = new Label
        {
            Text = "0",
            Font = Font.SystemFontOfSize(NamedSize.Large),
            XAlign = TextAlignment.End,
            YAlign = TextAlignment.Center
        };
        grid.Children.Add(displayLabel, 0, 2, 0, 1);

        backspaceButton = new Button
        {
            Text = "¥u21E6",
            Font = Font.SystemFontOfSize(NamedSize.Large),
            BorderWidth = 1,
            IsEnabled = false
        };
        backspaceButton.Clicked += OnBackspaceButtonClicked;
        grid.Children.Add(backspaceButton, 2, 0);

        // Now do the 10 number keys.
        for (int num = 0; num < 10; num++)
        {
            Button numberButton = new Button
            {
                Text = num.ToString(),
                Font = Font.SystemFontOfSize(NamedSize.Large),
                StyleId = num.ToString(),
                BorderWidth = 1
            };
            numberButton.Clicked += OnNumberButtonClicked;

            int left = (num == 0) ? 0 : (num + 2) % 3;
            int right = (num == 0) ? 3 : left + 1;
            int top = (12 - num) / 3;
            int bottom = top + 1;
            grid.Children.Add(numberButton, left, right, top, bottom);
        }

        this.Content = grid;
    }

    …
}
```

The two event handlers remain the same.

The code is considerably simplified, even with the calculations of `left`, `right`, `top`, and `bottom` to create a bottom-up calculator keyboard rather than a top-down telephone keyboard. The earlier program required six instances of `StackLayout`; now there's only one `Grid`.

Notice that `HorizontalOptions` and `VerticalOptions` are set on the `Grid` to center it on the page, but these properties are not set on any of the children. `HorizontalOptions` and `VerticalOptions` have *no* effect on children of a `Grid`, but the `XAlign` and `YAlign` properties are set on the `Label` to vertically center the text and keep it at the right:

Three side-by-side screenshots showing the KeypadGrid program displaying a numeric keypad running on iPhone, Android, and Windows Phone.

Unlike the horizontal `StackLayout` in a vertical `StackLayout`, the `Grid` has put the buttons in nice straight columns.

However, it's not exactly correct: In a keypad, you probably want all the keys to be the same size, but they're not in this program. The problem is particularly noticeable on the Windows Phone: The third column is wider than the middle column because the width of each column is based on the widest view in that column, and in the default Windows Phone font, the 6 and 9 are wider than other digits.

By default, the sizes of the columns and rows of a `Grid` are based on the sizes of the child views. There are two alternatives: You can give each column width or row height a specific size in device-independent units. Or, you can allocate space proportionally among the cells.

For either of these alternatives, you need to make use of the `ColumnDefinitions` and `RowDefinitions` properties defined by `Grid`. These are collections of `ColumnDefinition` and `RowDefini-tion` objects, respectively, each of which indicates a column width or row height.

`ColumnDefinition` has a property named `Width` and `RowDefinition` has a property named `Height`, and you set both to a value of type `GridLength`. You have three options, and you can use a

different option for each column or row of the `Grid`:

- `Auto`: This is the default. The column width or row height is based on the child width and height.
- `Absolute`: A column width or row height in device-independent units.
- `Star`: Similar to the star (asterisk) option in HTML tables, which allocates the space proportionally.

If there is no `ColumnDefinition` or `RowDefinition` object corresponding to a particular column or row, `Auto` is used.

Try inserting this code in the `KeypadGridPage` constructor any time after the `Grid` has been created:

```
for (int col = 0; col < 3; col++)
{
    grid.ColumnDefinitions.Add(new ColumnDefinition
        {
            Width = new GridLength(1, GridUnitType.Star)
        });
}
```

Now all the columns have the same width:

The `Star` option allocates space proportionally based on the first argument to `GridLength`. In the above example, all three columns have the same relative width of 1, but if you instead set it to (col + 1):

```
for (int col = 0; col < 3; col++)
{
    grid.ColumnDefinitions.Add(new ColumnDefinition
        {
            Width = new GridLength(col + 1, GridUnitType.Star)
        });
}
```

then the second column will be twice as wide as the first, and the third column will be three times as wide.

You can specify a width or height in device-independent units (for example, 150) like this:

```
new GridLength(150, GridUnitType.Absolute)
```

Or with the single-argument constructor:

```
new GridLength(150)
```

You can set it to `Auto` using:

```
new GridLength(1, GridUnitType.Auto)
```

Or using a static property:

```
GridLength.Auto
```

During the layout process, the `Grid` first totals up all the `Absolute` and `Auto` sizes horizontally and vertically, and then allocates remaining space among all the `Star` widths and heights.

It is possible to create a single-dimension `Grid`. The `AddHorizontal` and `AddVertical` methods append children to the end of the row or column, and these methods also work with `Ienumerable-<View>` collections, such as arrays.

The Grid and attached bindable properties

Although the `Grid` redefines its `Children` property for `Add` methods that include cell indices, and for the `AddHorizontal` and `AddVertical` methods, you can also use the normal `Add` method:

```
grid.Children.Add (child);
```

At first this seems insufficient because it doesn't include arguments to indicate the column and row for this child. But like `AbsoluteLayout`, the `Grid` defines attached bindable properties that let you specify that information. You can set the column and row of a child of a `Grid` using these two static methods:

```
Grid.SetColumn (child, 2);
Grid.SetRow (child, 3);
```

`Grid` also defines static `Grid.SetColumnSpan` and `Grid.SetRowSpan` methods. The row and

column spans have default values of 1. `Grid` also defines static `Get` methods to obtain the current values. As with the static methods defined by the `AbsoluteLayout` class, these are implemented by referencing attached bindable properties. The `Grid.SetColumn` and `Grid.SetRow` calls are equivalent to the following calls on the child view object:

```
child.SetValue (Grid.ColumnProperty, 2);
child.SetValue (Grid.RowProperty, 3);
```

That is how the static properties are defined.

There is no difference in calling

```
grid.Children.Add (child, 2, 3);
```

and calling:

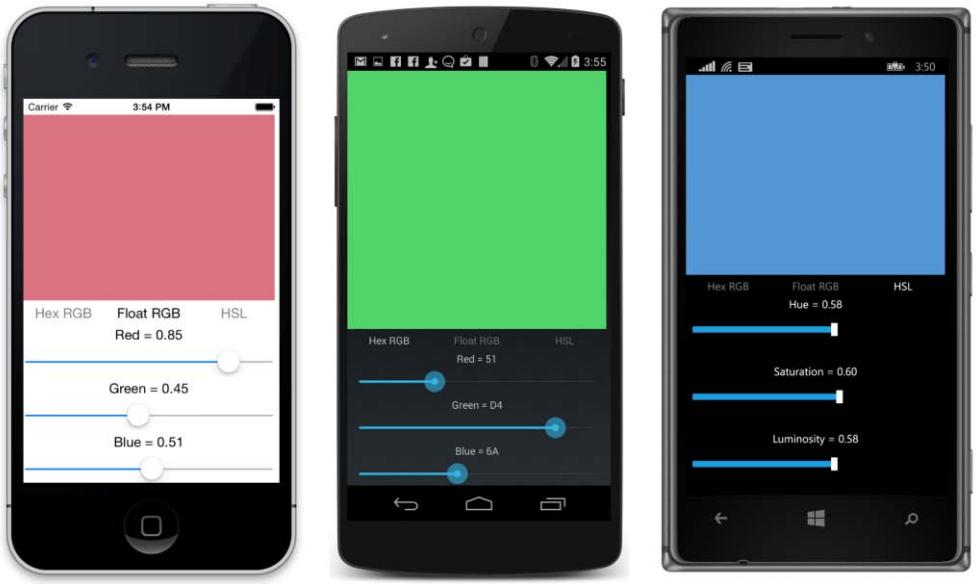
```
Grid.SetColumn (child, 2);
Grid.SetRow (child, 3);
grid.Children.Add (child);
```

The shortcut methods are implemented using the static methods defined by `Grid` and the `Add` call.

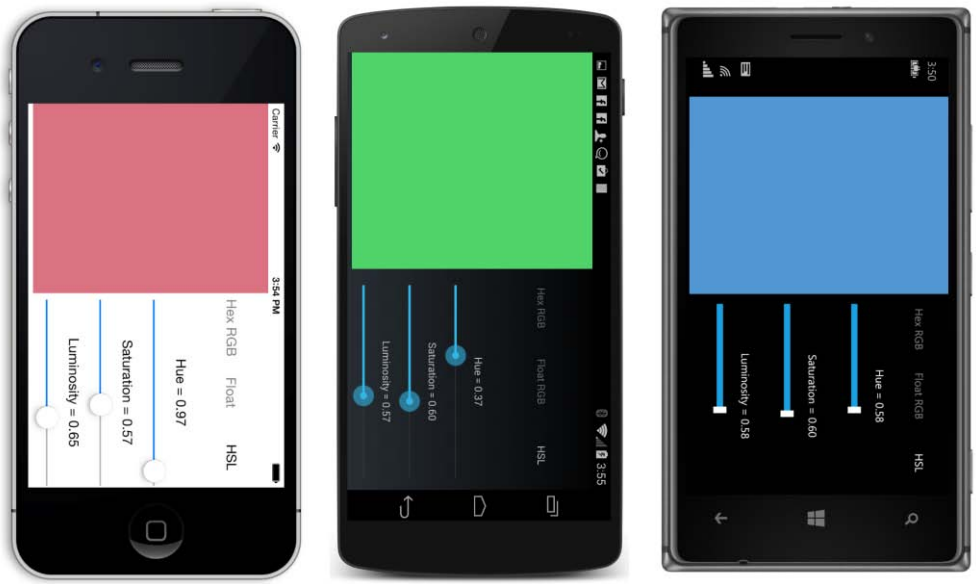
In many cases, when you define `Grid` contents using the extended `Add` methods, you don't need to use these static methods or bother with the `Grid` attached bindable properties. However, sometimes they come in very handy to change the rows and columns of `Grid` contents at a later time. For example, shifting children around in a Grid is one way to adapt your page to orientation changes.

Here's a program called **ColorScroll** that consists of a "control panel" of sorts with three `Label` views used as radio buttons, and three labeled `Slider` views for selecting a color. The resultant color is displayed by a `BoxView` that occupies half the page.

In portrait mode, the control panel is on the bottom and the `BoxView` is on the top:



In landscape mode, the control panel is on the right and the `BoxView` is on the left:



This accommodation of orientation is implemented with a 2 × 2 `Grid`. In portrait mode, the width of the second column is set to zero and the `BoxView` and control panel are positioned in rows 0 and 1 of the first column. In landscape mode, code in the `SizeChanged` event handler for the page sets the height of the second row to zero and repositions the `BoxView` and control panel to columns 0 and 1 of the first row.

ColorScroll has some flexibility of another sort: It lets you specify a color with the sliders in hexadecimal RGB values, floating-point RGB values, or HSL values. The three `Label` views at the top of the control panel use the `RadioExtensions` class shown earlier; unselected items have their `Opacity` properties set to 0.5. Many of the fields in the `ColorScrollPage` class—including the `ColorMode` enumeration and the three arrays of labels—exist to support the switching among these three color representations:

```
class ColorScrollPage : ContentPage
{
    Grid mainGrid;
    StackLayout controllersStack;
    BoxView boxView;
    Label[] labels = new Label[3];
    Slider[] sliders = new Slider[3];

    enum ColorMode
    {
        RgbHex,
        RgbFloat,
        Hsl
    }

    ColorMode currentColorMode = ColorMode.RgbHex;
    Color currentColor = Color.Aqua;
    bool ignoreSliderValueChanges;

    string[] rgbHexFormat = { "Red = {0:X2}", "Green = {0:X2}", "Blue = {0:X2}" };
    string[] rgbFloatFormat = { "Red = {0:F2}", "Green = {0:F2}", "Blue = {0:F2}" };
    string[] hslFormat = { "Hue = {0:F2}", "Saturation = {0:F2}", "Luminosity = {0:F2}" };

    …
}
```

As usual, the constructor performs much of the layout. There are two `Grid` layouts. The one called `mainGrid` is the 2 × 2 `Grid` that occupies the entire page; one cell contains the `StackLayout` with all the views for controlling the color, including a three-column `Grid` with the three radio button labels. To make it stand out a bit, this three-column `Grid` has its `VerticalOptions` property set to `LayoutOptions.CenterAndExpand` to use all the extra vertical space.

```
class ColorScrollPage : ContentPage
{

    …

    public ColorScrollPage()
    {
        // Define a 2 x 2 Grid that will be modified for portrait
        //      or landscape in SizeChanged handler.
        mainGrid = new Grid
        {
            ColumnDefinitions =
            {
```

```
            new ColumnDefinition(),
            new ColumnDefinition()
        },
        RowDefinitions =
        {
            new RowDefinition(),
            new RowDefinition()
        }
    };

    // Put all Labels and Sliders in StackLayout.
    // Wait until SizeChanged to set row and column.
    controllersStack = new StackLayout();
    mainGrid.Children.Add(controllersStack);

    // Also wait until SizeChanged to set BoxView row and column.
    boxView = new BoxView
    {
        Color = currentColor
    };
    mainGrid.Children.Add(boxView);

    // Assemble 3-column Grid to display color options.
    Grid radioGrid = new Grid
    {
        VerticalOptions = LayoutOptions.CenterAndExpand,
    };

    string[] modeLabels = { "Hex RGB", "Float RGB", "HSL" };

    foreach (ColorMode colorMode in Enum.GetValues(typeof(ColorMode)))
    {
        // Define a Label to be used as a radio button.
        Label radioLabel = new Label
        {
            Text = modeLabels[(int)colorMode],
            StyleId = colorMode.ToString(),
            XAlign = TextAlignment.Center,
            Opacity = 0.5
        };
        radioLabel.AddRadioToggler(OnRadioLabelToggled);

        // Set default item.
        radioLabel.SetRadioState(colorMode == ColorMode.RgbHex);

        // Add it to the Grid.
        radioGrid.Children.AddHorizontal(radioLabel);

        // Set the column width to "star" to equally space them.
        radioGrid.ColumnDefinitions.Add(new ColumnDefinition
        {
            Width = new GridLength(1, GridUnitType.Star)
        });
    }
```

```
        controllersStack.Children.Add(radioGrid);

        // Create Labels and Sliders for the three color components.
        for (int component = 0; component < 3; component++)
        {
            labels[component] = new Label
            {
                XAlign = TextAlignment.Center
            };
            controllersStack.Children.Add(labels[component]);

            // Set same ValueChanged handler for all sliders.
            sliders[component] = new Slider();
            sliders[component].ValueChanged += OnSliderValueChanged;
            controllersStack.Children.Add(sliders[component]);
        }

        // Build page.
        this.Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
        this.Content = mainGrid;

        // Set SizeChanged handler.
        SizeChanged += OnPageSizeChanged;

        // Initialize Slider values.
        SetSlidersAndBindings ();
    }

    …
}
```

The `SizeChanged` handler for the page is responsible for manipulating the row and column definitions of the `mainGrid` depending on the aspect ratio, and for putting the `BoxView` and the `StackLayout` with the control panel in the appropriate cells. Although some of these settings are the same regardless of the orientation—and the column and row are zero by default—defining and redefining them all in one place clarifies the intent:

```
class ColorScrollPage : ContentPage
{

    …

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // Portrait mode.
        if (this.Height > this.Width)
        {
            // Adjust column and row definitions.
            mainGrid.ColumnDefinitions[0].Width = new GridLength(1, GridUnitType.Star);
            mainGrid.ColumnDefinitions[1].Width = new GridLength(0);
            mainGrid.RowDefinitions[0].Height = new GridLength(1, GridUnitType.Star);
            mainGrid.RowDefinitions[1].Height = GridLength.Auto;
```

```
            // Set rows and columns of BoxView and StackLayout.
            Grid.SetColumn(boxView, 0);
            Grid.SetRow(boxView, 0);

            Grid.SetColumn(controllersStack, 0);
            Grid.SetRow(controllersStack, 1);
        }
        // Landscape mode.
        else
        {
            // Adjust column and row definitions.
            mainGrid.ColumnDefinitions[0].Width = new GridLength(1, GridUnitType.Star);
            mainGrid.ColumnDefinitions[1].Width = new GridLength(1, GridUnitType.Star);
            mainGrid.RowDefinitions[0].Height = new GridLength(1, GridUnitType.Star);
            mainGrid.RowDefinitions[1].Height = new GridLength(0);

            // Set rows and columns of BoxView and StackLayout.
            Grid.SetColumn(boxView, 0);
            Grid.SetRow(boxView, 0);

            Grid.SetColumn(controllersStack, 1);
            Grid.SetRow(controllersStack, 0);
        }
    }

    …

}
```

The event handler for the three `Label` views that function as radio buttons has two jobs: to set the `Opacity` of the `Label` to indicate which has been selected, and to set a new color mode. However, the job of actually making all the changes to the sliders and labels as a result is relegated to a separate method:

```
class ColorScrollPage : ContentPage
{

    …

    void OnRadioLabelToggled(View view)
    {
        // Set Opacity to indicate toggled state.
        view.Opacity = view.GetRadioState() ? 1.0 : 0.5;

        // Possibly change the current color mode.
        if (view.GetRadioState ())
        {
            ColorMode newColorMode =
                (ColorMode) Enum.Parse (typeof(ColorMode), view.StyleId);

            if (currentColorMode != newColorMode)
            {
                currentColorMode = newColorMode;
```

```
                SetSlidersAndBindings ();
            }
        }
    }

    …

}
```

The `SetSlidersAndBindings` method has the responsibility of setting all the `Slider` properties in accordance with the new color mode. For example, for the hexadecimal RGB mode, the `Maximum` property of each `Slider` must be set to 255. Based on the color mode, the `Slider` values themselves must be set based on the `R`, `G`, and `B`, or the `Hue`, `Saturation`, and `Luminosity` properties of the current color.

`SetSlidersAndBindings` is also responsible for setting bindings from the `Value` properties of the three `Slider` views to target the three `Label` views that identify these sliders. These bindings use the groups of formatting strings defined as field arrays. For the floating point RGB or HSL mode, the formatting specification is "F2", and for the hexadecimal RGB mode, the formatting specification is "X2."

And that is a problem. The `Value` property of the `Slider` is of type `double`, and if you attempt to format a `double` with "X2", an exception will be raised.

In code, converting from a `double` to an `int` is a trivial cast. But when defining a data binding, you don't have direct access to the data passing from the source to the target. So how can this be done?

Fortunately, the `Binding` constructor and the `Binding.Create` method have an argument of type `IValueConverter` specifically for this purpose. `IValueConverter` is an interface that defines two methods named `Convert` and `ConvertBack`. You supply a class that implements this interface to perform the conversion you need. The `Convert` method is called when the property value passes from the source to the target. The `ConvertBack` method only plays a role in `TwoWay` or `OneWayToSource` bindings to convert the target type to the source type.

Here is an implementation of `IValueConverter` to convert from `double` to `int`:

```
public class DoubleToIntegerConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        return (int)(double)value;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, CultureInfo culture)
    {
        return (double)(int)value;
    }
}
```

Although this is certainly one of the simplest `IValueConverter` implementations imaginable, they often aren't very elaborate. In `Convert`, the `value` argument is the value from the source, and the return value goes to the target. You can customize the method by checking the `targetType` argument, which is the type of the target property. You can also pass a `parameter` from the `Binding` constructor or `Binding.Create` method. You'll see more examples in future chapters.

The `SetSlidersAndBindings` method instantiates `DoubleToIntConverter` when it's required for the bindings. In a real program, you only need one instance of `DoubleToIntConverter` for the lifetime of the program, and you can instantiate it as a field.

```
class ColorScrollPage : ContentPage
{

    …

    void SetSlidersAndBindings()
    {
        // Don't handle ValueChange events from Sliders!
        ignoreSliderValueChanges = true;

        double sliderMax = 1;
        string[] format = null;
        IValueConverter valueConverter = null;

        // Set local variables for color mode.
        switch (currentColorMode)
        {
            case ColorMode.RgbHex:
                sliderMax = 255;
                format = rgbHexFormat;
                valueConverter = new DoubleToIntegerConverter();
                break;

            case ColorMode.RgbFloat:
                format = rgbFloatFormat;
                break;

            case ColorMode.Hsl:
                format = hslFormat;
                break;
        }

        // Bind the labels to the sliders.
        for (int i = 0; i < 3; i++)
        {
            sliders[i].Maximum = sliderMax;
            labels[i].BindingContext = sliders[i];
            labels[i].SetBinding(Label.TextProperty,
                new Binding("Value", BindingMode.OneWay, valueConverter, null, format[i]));
        }

        // Now set the slider values.
        Color currentColor = this.currentColor;
```

```
        switch (currentColorMode)
        {
            case ColorMode.RgbHex:
                sliders[0].Value = 255 * currentColor.R;
                sliders[1].Value = 255 * currentColor.G;
                sliders[2].Value = 255 * currentColor.B;
                break;

            case ColorMode.RgbFloat:
                sliders[0].Value = currentColor.R;
                sliders[1].Value = currentColor.G;
                sliders[2].Value = currentColor.B;
                break;

            case ColorMode.Hsl:
                sliders[0].Value = currentColor.Hue;
                sliders[1].Value = currentColor.Saturation;
                sliders[2].Value = currentColor.Luminosity;
                break;
        }

        // Resume handling ValueChange events.
        ignoreSliderValueChanges = false;

        // Sync up the colors and sliders.
        SetColorFromSliders();
    }
    …

}
```

The `SetSlidersAndBindings` method manually sets the `Value` property of each `Slider`, and this triggers the `ValueChanged` event, which then (as you can see below) sets the color from the `Slider` values. This means that the `currentColor` changes as it's being accessed to set the `Slider` values. The purpose of the `ignoreSliderValueChanges` field is to prevent those recursive calls.

```
class ColorScrollPage : ContentPage
{

    …

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (ignoreSliderValueChanges)
            return;

        SetColorFromSliders();
    }

    void SetColorFromSliders()
    {
```

```
            double d0 = sliders[0].Value;
            double d1 = sliders[1].Value;
            double d2 = sliders[2].Value;

            switch (currentColorMode)
            {
                case ColorMode.RgbHex:
                    currentColor = Color.FromRgb((int)d0, (int)d1, (int)d2);
                    break;

                case ColorMode.RgbFloat:
                    currentColor = Color.FromRgb(d0, d1, d2);
                    break;

                case ColorMode.Hsl:
                    currentColor = Color.FromHsla(d0, d1, d2);
                    break;
            }
            boxView.Color = currentColor;
        }
}
```

This method is called when you switch between the three color modes, but the color itself does not change.

Despite the data bindings between the sliders and labels, this **ColorScroll** program is rather complex and has a lot of moving parts. Can it be simplified?

Almost certainly it can. One way is to extract all the color-calculation logic, and put it in a separate class that implements `INotifyPropertyChanged`, and then define data bindings between that class and the sliders and `BoxView` that displays the resultant color. This is classic MVVM (Model-View-ViewModel) architecture: The user interface is the View, which interacts through data bindings with the ViewModel—the class that implements `INotifyPropertyChanged`. If necessary, the ViewModel can make use of an underlying data Model, and in this case, the Model is the Xamarin.Forms `Color` structure itself.

Another way to simplify this program is to convert the user interface from code to markup. Xamarin.Forms supports XAML, the eXtensible Application Markup Language made popular in several Microsoft platforms. XAML often imposes a clean separation between user-interface objects and underlying logic, but something else very interesting often happens when defining a user-interface in XAML: You begin to see ways to build code that supports the markup in very structured ways.

It might seem a little strange, but using XAML can make you a better C# programmer!

And that means that this exploration of Xamarin.Forms is only just beginning.

From technical overviews to drilldowns on special topics, get *free* ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press

Now that you've read the book...

Tell us what you think!

Was it useful?
Did it teach you what you wanted to learn?
Was there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!