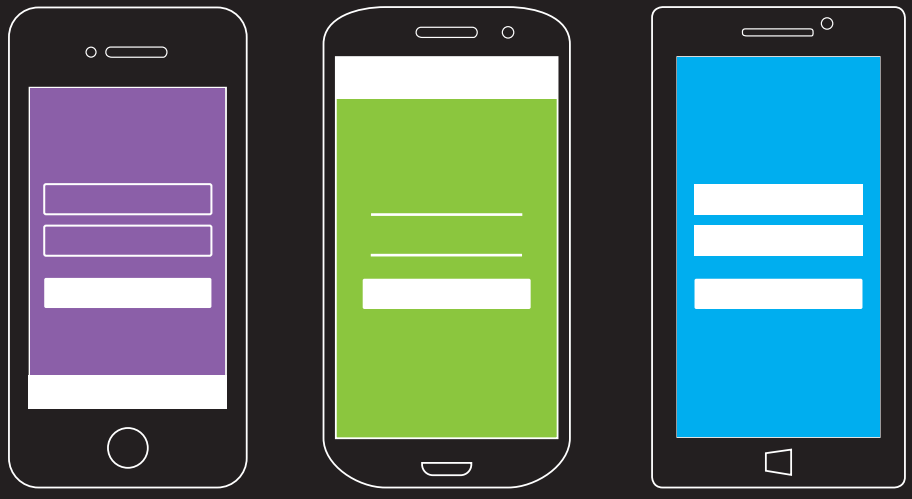
Xamarin

Microsoft

Creating Mobile Apps with Xamarin.Forms



PREVIEW EDITION 2

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: 978-0-7356-9723-2

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
Editorial production: John Pierce, Flying Squirrel Press
Cover illustration: Serena Zhang

Table of contents

Introduction

This is the second 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 16 chapters and doesn't cover some important topics. The final edition of the book will probably be published in the summer of 2015.

All information about this book can be found on the book's home page at:

http://developer.xamarin.com/guides/cross-platform/xamarin-forms/creating-mobile-apps-xamarin-forms/

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 programmers started with these platforms or in constructing a prototype or proof-of-concept application.

This book assumes that you know C# and are familiar 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.

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 **MonkeyTap**.

The 1st and 2nd Preview Editions

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 at the Xamarin.Forms Framework link on this page: http://api.xamarin.com/.

The first Preview Edition of this book was published in October 2014 to coincide with the Xamarin Evolve conference. It contained six chapters but no coverage of XAML.

This second Preview Edition was reconceived to contain shorter and more focused chapters. After the book establishes a solid foundation in code-only Xamarin.Forms programming, XAML is introduced in Chapter 7, and from that point on, XAML is used to define the user interface of most of the sample programs.

This Preview Edition unfortunately ends without covering some essential topics. The most obvious missing topics include the `Grid`, collection views such as `ListView`, page navigation, triggers, behaviors, and maps. However, it is our intention to publish chapters online just about as quickly as I can write them. New chapters will appear on the Xamarin webpage devoted to this book.

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 later (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 are 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.

All the screen shots in this book show an iPhone, an Android phone, and a Windows Phone in that order. The three devices shown in these screen shots reflect my setup and hardware:

- The iPhone 6 simulator on the MacBook Pro running iOS 8.2

- An LG Nexus 5 running Android 5.1.
- A Nokia Lumia 925 running Windows Phone 8.1

Some of the early screen shots in this book were from devices with somewhat earlier versions of the operating systems, for example Android 5.0 or 5.01 rather than 5.1.

The Xamarin.Forms programs in this book target the Xamarin.iOS Unified API and the Windows Phone 8.0 Silverlight API. (More about the Windows Phone API shortly.)

Downloads: Code samples

The sample programs shown in the pages of this book were compiled in late March 2015 with version 1.4.0 of Xamarin.Forms. The source code of these samples is hosted on a repository on GitHub: https://github.com/xamarin/xamarin-forms-book-preview-2/.

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
- Xamarin.Forms.Platform.WP8.dll.

The Xamarin.Forms package also requires:

- Xamarin Support Library v4 (Xamarin.Android.Support.v4.dll)
- Windows Phone Toolkit (Microsoft.Phone.Controls.Toolkit.dll)

These 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 to 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 select **Updates** and **nuget.org** at the left to choose to update the package. If an update is available, clicking the **Update All** button is the easiest way 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 drop-down.

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 by default 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.

Big changes coming for Windows Phone

As this second Preview Edition is being finalized in late March 2015, a big change for Xamarin.Forms is just becoming available in a preview release: this is support for the Windows Runtime API supported by Windows Phone 8.1 and by Windows 8.0 Windows Store applications.

The contents of this book do not reflect that change! All the programs in this book target the Silverlight API of Windows Phone 8.0. Most of these programs should migrate easily to the Windows Runtime API. However, any program that relies upon Windows Phone 8.0 platform services—in particular, the programs in Chapter 9, "Platform-specific API calls"—will need some work.

By the time you see this book, some of the work converting the sample programs to the Windows Runtime platform should already have been completed. It is anticipated that the GitHub repository with the sample code for this book will have a branch named **windows-runtime** with the converted samples.

Acknowledgments

It's always seemed peculiar to me that authors of programming books 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, Seth Rosetter, Rui Marinho, and Chris King. Congratulations, guys! We've been enjoying the fruits of your labor!

Over the months that the first and second Preview Editions were 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 managed to persuade me to take a somewhat different approach to the material. Later on, Craig kept me on track and reviewed the chapters in this second Preview Edition, and John Meade did the copyediting. For the first Preview Edition, Stephane Delcroix at Xamarin and Andy Wigley with Microsoft offered essential technical reads and persistently prodded me to make the book better. Microsoft's copyeditor for this second Preview Edition was John Pierce.

Almost nothing I do these days would be possible without the daily companionship and support of my wife, Deirdre Sinnott.

Charles Petzold
March 23, 2015

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 2

Anatomy of an app

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 between multiple pages. In many of the early chapters in this book, 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, in a two-dimensional grid, or in a more freeform manner. In this chapter, however, our pages will contain just a single 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. This book refers to them as views or elements. In this chapter, you'll encounter the `Label` view for displaying text.

Say hello

Using either Microsoft Visual Studio or Xamarin Studio, let's create a new Xamarin.Forms application by

using a standard template. This process creates a solution that contains up to four projects: three platform projects—for iOS, 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 section of the dialog lists three 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, pick the first one: **Blank App (Xamarin.Forms Portable)**. 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 (the PCL project) and three application projects. For a solution named **Hello**, these are:

- A Portable Class Library project named **Hello** that is referenced by the three application projects;
- An application project for Android, named **Hello.Droid**;
- 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, neither the iOS nor the Windows Phone program is created.

Before continuing, check to be sure that the project configurations are okay. In Visual Studio, select the **Build > Configuration Manager** menu item. In the **Configuration Manager** dialog, you'll see the PCL project and the three application projects. Make sure the **Build** box is checked for all the projects

and the **Deploy** box is checked for all the application projects (unless the box is grayed out). Take note of the **Platform** column: If the **Hello** project is listed, it should be flagged as **Any CPU**. The **Hello.Droid** 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 on 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.

If a project doesn't seem to be compiling or deploying in Visual Studio, recheck the settings in the **Configuration Manager** dialog. Sometimes a different configuration becomes active and might not include the PCL project.

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

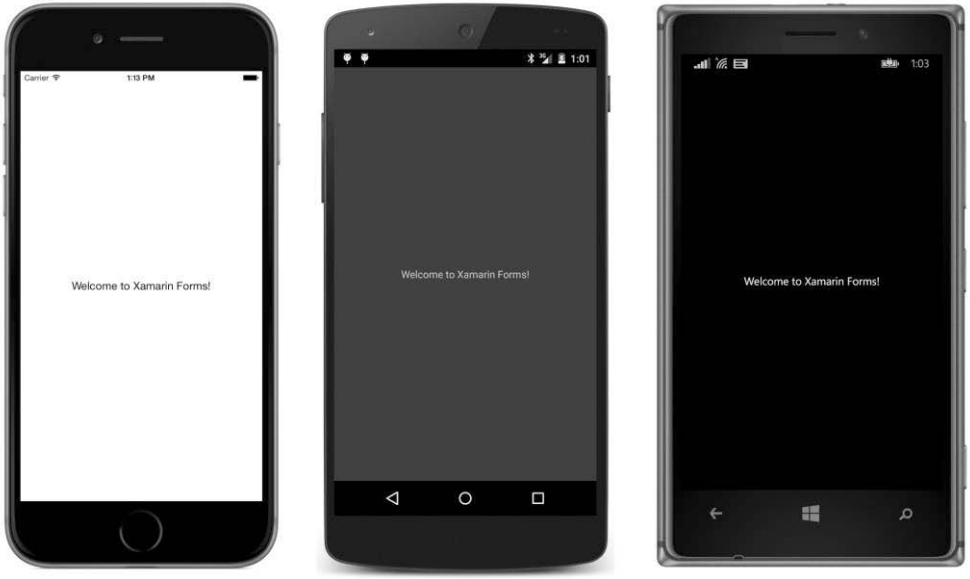
In Visual Studio, you'll probably want to display the iOS and Android toolbars. These toolbars let you choose among emulators and devices and allow you to manage the emulators. From the main menu, make sure the **View > Toolbars > iOS** and **View > Toolbars > Android** items are checked.

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 three application projects and select the **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 tool 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 the Android color scheme in that it displays light text on a dark background—but on Windows Phone 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 appears 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 displays, you can change that in the individual platform projects.

Inside the files

Clearly, 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` class in the **Hello** project. In a project created by Visual Studio, the `App` class is defined in the App.cs file, but in Xamarin Studio, the file is Hello.cs. 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 : Application
    {
        public App()
        {
            // The root page of your application
            MainPage = new ContentPage
            {
                Content = new StackLayout
                {
                    VerticalOptions = LayoutOptions.Center,
                    Children = {
                        new Label {
                            XAlign = TextAlignment.Center,
                            Text = "Welcome to Xamarin Forms!"
                        }
                    }
                }
            };
        }

        protected override void OnStart()
        {
            // Handle when your app starts
        }

        protected override void OnSleep()
        {
            // Handle when your app sleeps
        }

        protected override void OnResume()
        {
            // Handle when your app resumes
        }
    }
}
```

Notice that the namespace is the same as the project name. This `App` class is defined as public and derives from the Xamarin.Forms `Application` class. The constructor really has just one responsibility: to set the `MainPage` property of the `Application` class to an object of type `Page`.

The code that the Xamarin.Forms template has generated here shows one very simple approach to defining this constructor: 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 the status bar at the top of the Android screen, the buttons on the bottom of the Android screen, and the status bar at the top of the Windows Phone screen. (As you'll discover, the iOS status bar is actually part of the `ContentPage`.)

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, and in this case it's set to a `StackLayout`, which arranges its children in a stack.

This `StackLayout` has only one child, which is a `Label`. The `Label` class derives from `View` and is used in Xamarin.Forms applications to display up to a paragraph of text. The `VerticalOptions` and `XAlign` 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 constructor of the `App` class then sets an instance of the class that you define to its `MainPage` property. You'll see how this works shortly.

In the **Hello** solution, you'll also see an AssemblyInfo.cs file for creating the PCL and a packages.config file that contains the NuGet packages required by the program. In the **References** section under **Hello** in the solution list, you'll see the three libraries this PCL requires:

- .NET (displayed as .NET Portable Subset in Xamarin Studio)
- Xamarin.Forms.Core
- Xamarin.Forms.Xaml

It is this PCL 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 particular attention to these assets 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 PCL 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 `Forms.Init` method in the `Xamarin.Forms` namespace that initializes the Xamarin.Forms system for that particular platform. The startup code in each platform must make a call to this method.

You've also just seen that the PCL project derives a public class named `App` that derives from `Application`. The startup code in each platform must also instantiate this `App` class.

If you're familiar with iOS, Android, or Windows Phone development, you might be curious to see how the platform startup code handles these jobs.

The iOS Project

An iOS project typically contains a class that derives from `UIApplicationDelegate`. However, the Xamarin.Forms.Platform.iOS library defines an alternative base class named `FormsApplicationDelegate`. In the **Hello.iOS** project, you'll see this AppDelegate.cs file, here stripped of all extraneous `using` directives and comments:

```
using Foundation;
using UIKit;

namespace Hello.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate :
                            global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        public override bool FinishedLaunching(UIApplication app, NSDictionary options)
        {
            global::Xamarin.Forms.Forms.Init();
            LoadApplication(new App());

            return base.FinishedLaunching(app, options);
        }
    }
}
```

The `FinishedLaunching` override begins by calling the `Forms.Init` method defined in the **Xamarin.Forms.Platform.iOS** assembly. It then calls a `LoadApplication` method (defined by the `FormsApplicationDelegate`), passing to it a new instance of the `App` class defined in the shared PCL. The page object set to the `MainPage` property of this `App` object can then be used to create an object of type `UIViewController`, which 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 `FormsApplicationActivity` 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;

namespace Hello.Droid
```

```
{
    [Activity(Label = "Hello", Icon = "@drawable/icon", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsApplicationActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}
```

The new instance of the `App` class is then passed to a `LoadApplication` method defined by `FormsApplicationActivity`. The attribute set on the `MainActivity` class indicates that the activity is *not* re-created when the phone changes orientation (from portrait to landscape or back) or the screen changes size.

The Windows Phone project

In the Windows Phone project, look at the MainPage.xaml.cs file tucked underneath the Main-Page.xaml file in the project file list. This file defines the customary `MainPage` class, but notice that it derives from a Xamarin.Forms class named `FormsApplicationPage`. The newly instantiated `App` class is passed to the `LoadApplication` method defined by this base class:

```
using Microsoft.Phone.Controls;

namespace Hello.WinPhone
{
    public partial class MainPage : global::Xamarin.Forms.Platform.WinPhone.FormsApplicationPage
    {
        public MainPage()
        {
            InitializeComponent();
            SupportedOrientations = SupportedPageOrientation.PortraitOrLandscape;

            global::Xamarin.Forms.Forms.Init();
            LoadApplication(new Hello.App());
        }
    }
}
```

The setting of the `SupportedOrientations` property allows the phone to respond to orientation changes between portrait and landscape. The iOS and Android projects are enabled for orientation changes by default as well, so you should be able to turn the phone or simulator side to side and see the text realigned in the center of the screen.

Nothing special!

If you've created a Xamarin.Forms solution under Visual Studio and don't want 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 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 Unified API (not Classic API) Xamarin.iOS project by selecting the iOS project **Universal** type and **Blank App** template. Create a Xamarin.Android project with the Android **Blank App** template, or a Windows Phone 8.1 Silverlight project by selecting **Store Apps**, then **Windows Phone Apps**, then **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 Phone 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), whereas the second creates a Shared Asset Project (SAP) consisting only of shared code files. The original **Hello** solution used the PCL template. Now let's create a second solution named **HelloSap** with the SAP template.

As you'll see, everything looks pretty much the same, except that the **HelloSap** project itself contains only one item: the App.cs file.

With both the PCL and SAP approaches, code is shared among 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 SAP approach, the common code files are effectively included with the three application projects at build time. By default, the SAP has only a single file named App.cs, but effectively it's as if this **HelloSap** 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 a 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 later in this book, 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 drop-down 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 iOS project (that's two underscores before and after) and `WINDOWS_PHONE` is defined for the Windows Phone project. You won't see anything for Android, but the identifier `__ANDROID__` is defined anyway, as well as multiple `__ANDROID_nn__` identifiers, where `nn` is each Android API level supported. Your shared code file can include blocks like this:

```
#if __IOS__
            // iOS specific code
#elif __ANDROID__
            // Android specific code
#elif WINDOWS_PHONE
            // Windows Phone 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
- Xamarin.iOS (Classic)

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

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 Chapter 9—Xamarin.Forms provides a class named `DependencyService` that allows you to access platform-specific code from the PCL in a methodical manner.

Most of the programs in this book use the PCL approach. This is the recommended approach for Xamarin.Forms and is preferred by many programmers who have been working with Xamarin.Forms for a while. However, the SAP approach is also supported and definitely has its advocates as well. Programs within these pages that demonstrate the SAP approach contain the letters **Sap** at the end of their names, such as the **HelloSap** program.

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 PCL 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 **Forms ContentPage**. (Watch out: There's also a **Forms ContentView** option. Don't pick that one!)

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 **Forms**, and in the central area, select **Forms**

ContentPage. (Watch out: There are also **Forms ContentView** and **Forms ContentPage Xaml** options. Don't pick those!)

In either case, give the new file a name of GreetingsPage.cs.

The GreetingsPage.cs file will be initialized with some skeleton code for a class named `GreetingsPage` that derives from `ContentPage`. Because `ContentPage` is in the `Xamarin.Forms` namespace, a `using` directive includes that namespace. The class is defined as `public`, but it need not be because it won't be directly accessed from outside the `Greetings` project.

Let's delete all of the code in the `GreetingsPage` constructor and most of the `using` directives, so the file looks something like this:

```
using System;
using Xamarin.Forms;

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

        }
    }
}
```

In the constructor of the `GreetingsPage` class, instantiate 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 set the `MainPage` property to an instance of this `GreetingsPage` class:

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

```
namespace Greetings
{
    public class App : Application
    {
        public App()
        {
            MainPage = new GreetingsPage();
        }

        protected override void OnStart()
        {
            // Handle when your app starts
        }

        protected override void OnSleep()
        {
            // Handle when your app sleeps
        }

        protected override void OnResume()
        {
            // Handle when your app resumes
        }
    }
}
```

It's easy to forget this step, and you'll be puzzled that your program seems to completely ignore your page class and still says "Welcome to Xamarin Forms!"

It is in 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 book, the class that derives from `ContentPage` will have a name that is the same as the application but with `Page` appended. That naming convention should help you identify the code listings in this book from just the class or constructor name without seeing the entire file. In most cases, the code snippets in the pages of this book 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
        {
```

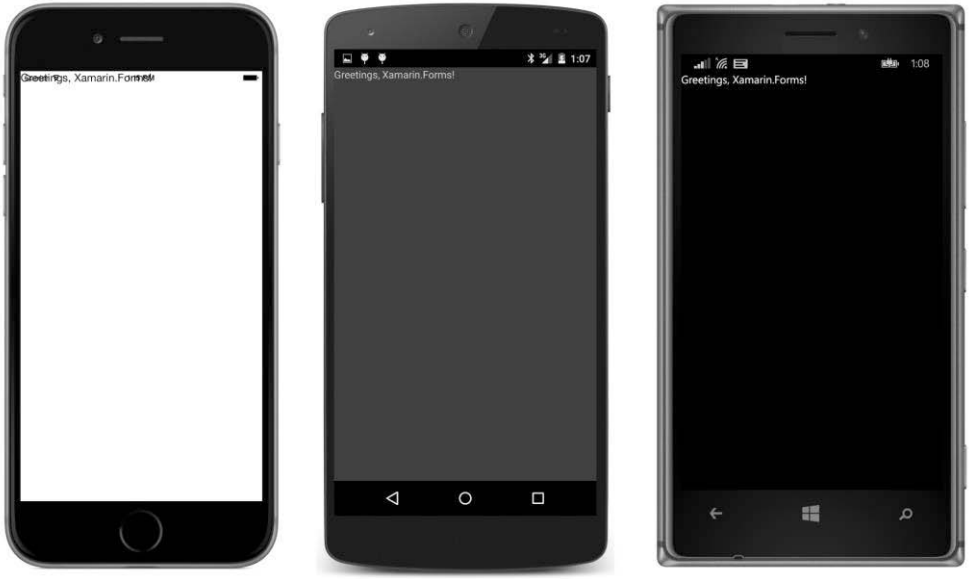
```
            Text = "Greetings, Xamarin.Forms!"
        };
        this.Content = label;
    }
}
```

This style allows the `Label` instance to be set to the `Content` property directly, so the `Label` doesn't require a name, like so:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        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 compile and run the program on the three platforms on either an emulator or a device, here's what you'll see:



The most disappointing version of this **Greetings** program is definitely the iPhone: Beginning 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 discussed later in this book, but until that time, here are four ways (or five ways if you're using an SAP) to solve this problem right away.

1. Include padding on the page

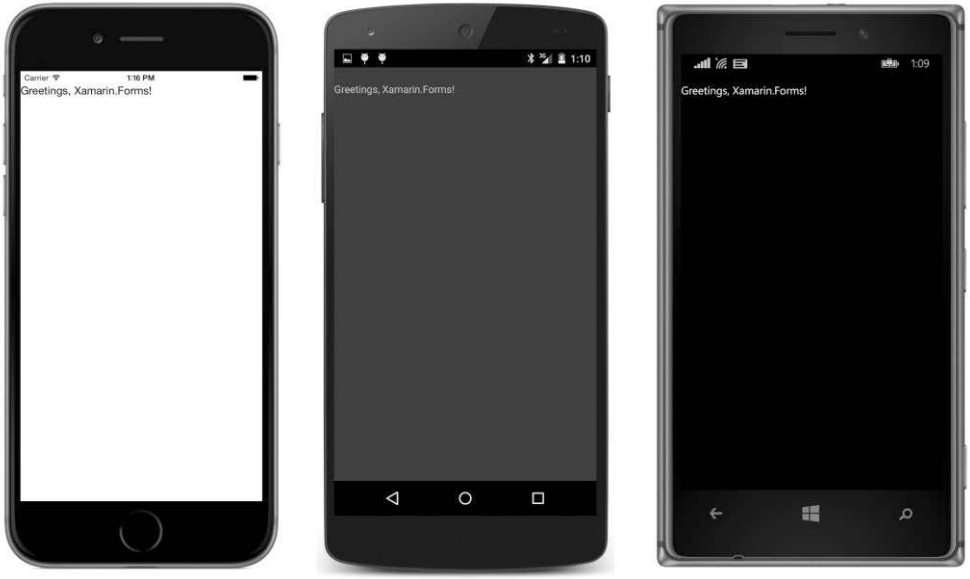
The `Page` class defines a property named `Padding` that marks an area around the interior perimeter of the page into which content cannot intrude. The `Padding` property is of type `Thickness`, a structure that defines four properties named `Left`, `Top`, `Right`, `Bottom`. (You might want to memorize that order because that's the order you'll define the properties 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 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, "Dealing with sizes," will provide some guidance when you need to get down to the pixel level.)

You can accommodate the status bar like so:

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

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

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



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?

2. Include padding just for iOS (Shared Asset Project only)

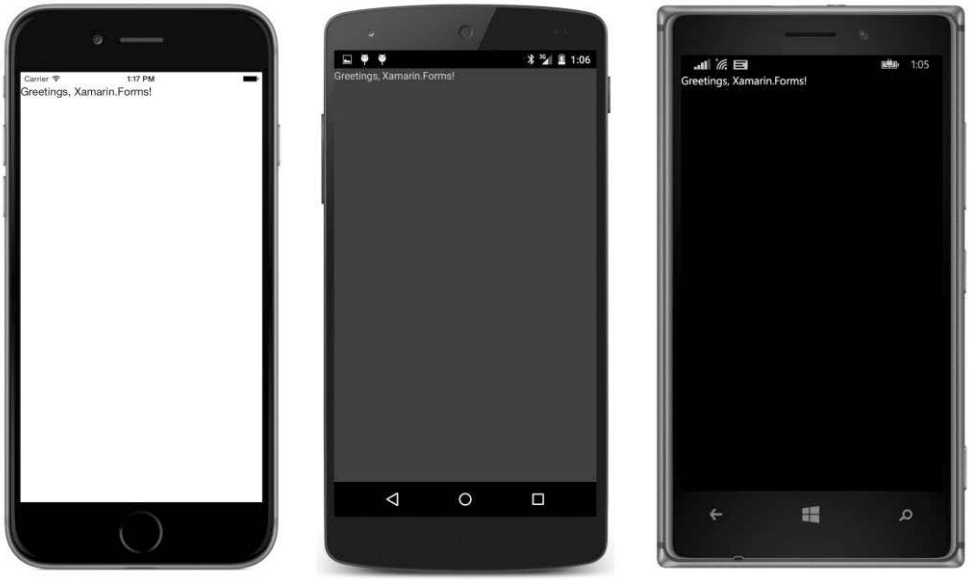
One of the advantages of the Shared Asset Project (SAP) approach is that the classes in the project are extensions of the application projects, so you can use conditional compilation directives.

Let's try this out. We'll need a new solution named **GreetingsSap** based on the SAP template, and a new page class in the **GreetingsSap** project named `GreetingsSapPage`. That class looks like this:

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

#if __IOS__

        Padding = new Thickness(0, 20, 0, 0);

#endif
        }
    }
```

```
}
```

The `#if` directive references the conditional compilation symbol `__IOS__`, so the `Padding` property is set only for the iOS project. The results look like this:



However, these conditional compilation symbols affect only the compilation of the program, so they have no effect in a PCL. Is there a way for a PCL project to include different `Padding` for the three platforms?

3. Include padding just for iOS (PCL or SAP)

Yes! The static `Device` class includes several properties and methods that allow your code to deal with device differences at run time 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`. (Although Xamarin.Forms is mostly intended for phones, you can certainly experiment with deploying to tablets.)

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 has some other purposes as well: It defines a static method to start running a timer and another to run some code on the main thread. This latter method comes in handy when you're working with asynchronous methods and supplemental threads because code that manipulates the user interface can generally be run only 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:

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

The first `Thickness` argument is for iOS, the second is for Android, and the third is for Windows Phone. 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, so this works as well:

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

Or, you can have just one `Thickness` constructor and use `Device.OnPlatform` for the second argument:

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

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

However, if you just need to set `Padding` for iOS, you can use the version of `Device.OnPlatform` with `Action` arguments. These arguments are `null` by default, so you can just set the first for an action to be performed on iOS:

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

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

```
            });
    }
}
```

Now the statement to set the padding is executed only when the program is running on iOS. Of course, with just that one argument to `Device.OnPlatform`, it could be a little obscure to people who need to read your code, so you might want to include the parameter name preceding the argument to make it explicit that this statement executes just for iOS:

```
Device.OnPlatform(iOS: () =>
    {
        Padding = new Thickness(0, 20, 0, 0);
    });
```

Naming the argument like that is a feature introduced in C# 4.0.

The `Device.OnPlatform` method is very handy and has the advantage of working in both PCL and SAP projects. However, it can't access APIs within the individual platforms. For that you'll need `DependencyService`, which is discussed in Chapter 9.

4. Center the label within the page

The problem with the text overlapping the iOS status bar occurs only 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 (in this case the `ContentPage`). These two properties are of type `LayoutOptions`, a structure with eight public static read-only fields that return `LayoutOptions` values:

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

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

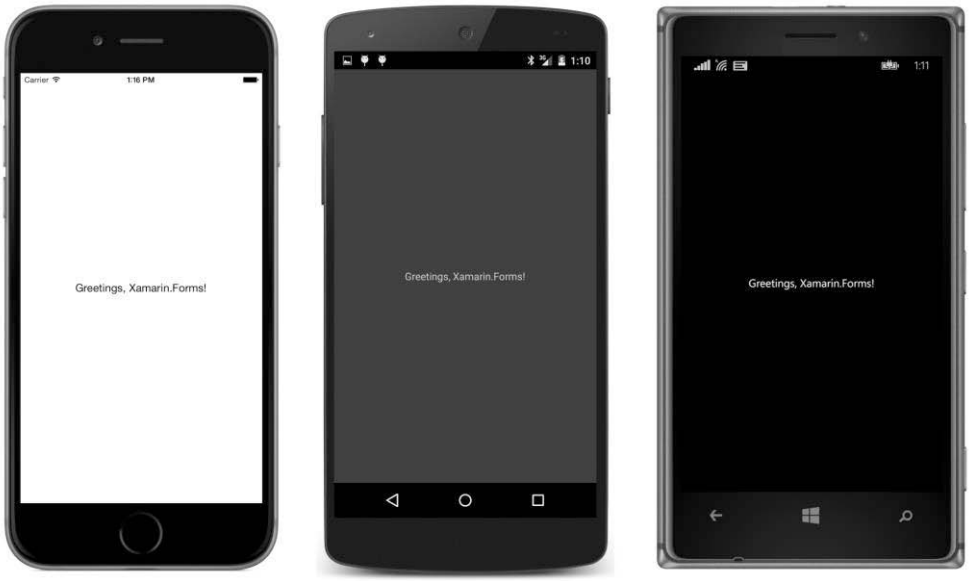
- An `Alignment` property of type `LayoutAlignment`, an enumeration with four members: `Start`, `Center`, `End`, and `Fill`.
- An `Expands` property of type `bool`.

A fuller explanation of how these work awaits you in Chapter 4, "Scrolling the stack," but for now you can set the `HorizontalOptions` and `VerticalOptions` properties of the `Label` to one of the static fields defined by `LayoutOptions` values. For `HorizontalOptions`, the word `Start` means left and `End` means right; for `VerticalOptions`, `Start` means top and `End` means bottom.

Mastering the use of the `HorizontalOptions` and `VerticalOptions` properties is a major part of acquiring skill in the Xamarin.Forms layout system, but here's a simple example that positions the `Label` in the center of the page:

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

5. 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 final solution to the iOS status bar problem, set `XAlign` and `YAlign` to `TextAlignment.Center`:

```
class GreetingsPage : ContentPage
{
    public GreetingsPage()
    {
        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 see in the next chapter.