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 occasions 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 while writing a program, including that there's a better way to structure the code. Sometimes, a partial or even a total rewrite can result in a much better application, or simply one that is structurally more coherent and easier to maintain. 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 a 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: an 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 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, a sense of dread pervades the forking of one application into two. With each line of code that 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 such a powerful allure 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 quick information, media consumption, and social networking. 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 for rapid change, currently two major phone and tablet platforms dominate:

- 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 currently in use, but iPhone and iPad users are more devoted and spend more time with their devices.

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

- Microsoft's Windows Phone and Windows 10 Mobile.

In recent years, these platforms have become a more compelling alternative as Microsoft has been merging the APIs of its mobile, tablet, and desktop platforms. Both Windows 8.1 and Windows Phone 8.1 are based on a single API called the Windows Runtime (or WinRT), which is based on Microsoft .NET. This single API means that applications targeted for desktop machines, laptops, tablets, and phones can share very much of their code.

Even more compelling is the Universal Windows Platform (UWP), a version of the Windows Runtime that forms the basis for Windows 10 and Windows 10 Mobile. A single UWP application can target every form factor from the desktop to the phone.

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 multitouch, but there are many differences in detail. 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 associated culture, 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, Android Studio on a variety of platforms.
- For Windows 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 a Boolean value:

- On the iPhone or iPad, it's a "view" called `UISwitch`.
- On Android devices, it's a "widget" called `Switch`.
- In the Windows Runtime API, it's a "control" called `ToggleSwitch`.

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

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 and iPad
- Java for Android devices
- C# for Windows

Objective-C, Java, and C# 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 wants 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#. Unveiled by Microsoft in the year 2000, C# 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 transformed C# so that it is now properly classified as a multiparadigm programming language. C# code can be traditionally imperative, or the code can be flavored with declarative or functional programming paradigms.

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 the extensive .NET Framework class library provides support 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 and JSON 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.

Soon after Microsoft's announcement of .NET way back 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 which has also adapted Mono to form the basis of cross-platform mobile solutions.

The year 2014 saw some developments in C# and .NET that bode well for its future. An open-source version of the C# compiler, called the .NET Compiler Platform (formerly known by its code name "Roslyn") has been published. And the .NET Foundation was announced to serve as a steward for open-source .NET technologies, in which Xamarin plays a major part.

In March 2016, Microsoft acquired Xamarin with the goal of bringing cross-platform mobile development to the wider Microsoft developer community. Xamarin.Forms is now freely available to all users of Visual Studio.

A single language for all platforms

For the first three years of its existence, Xamarin focused mainly on compiler technologies and three basic sets of .NET libraries:

- Xamarin.Mac, which has evolved from the MonoMac project.
- Xamarin.iOS, which evolved from MonoTouch.
- Xamarin.Android, which evolved from 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.

Developers can use Visual Studio to build Xamarin applications, targeting iOS and Android as well as all the various Windows platforms. However, iPhone and iPad development also requires a Mac connected to the PC through a local network. This Mac must have Xcode installed as well as Xamarin Studio, an OS X–based integrated development environment that lets you develop iPhone, iPad, Mac OS X, and Android applications on the Mac. Xamarin Studio does not allow you to target Windows 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, an application 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 a programmer develops an application 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.

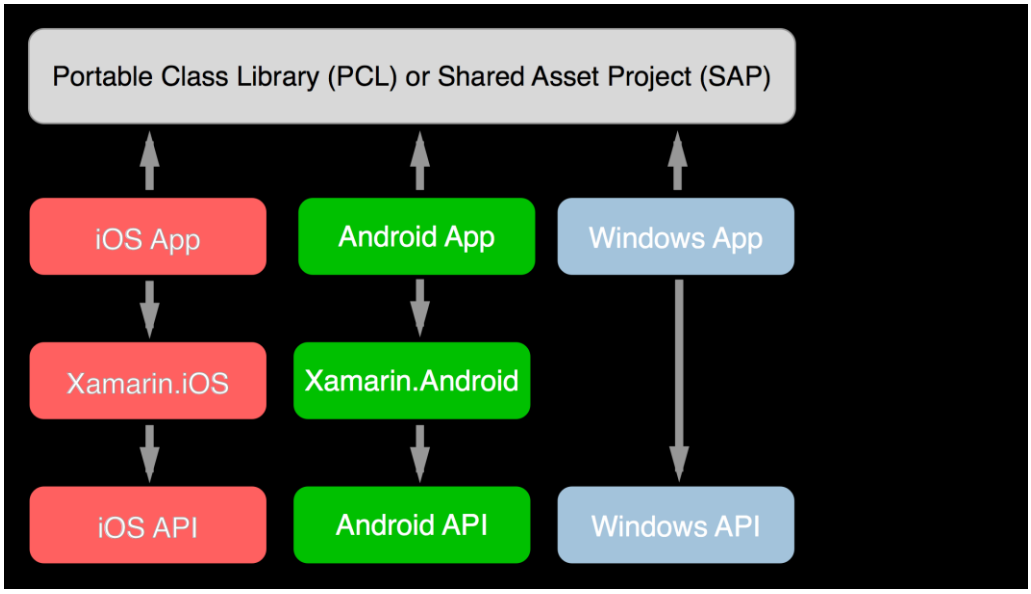
Often this platform-independent code needs to access files or the network or use collections or threading. Normally these jobs would be considered part of an operating system API, but they are also jobs that can make use of the .NET Framework class library, and if .NET is available on each platform, then this code 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 you use, 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 (all with access to a common PCL or SAP), or you can use Xamarin Studio to target iPhone and Android devices.

The following diagram illustrates the interrelationships between the Visual Studio or Xamarin Studio projects, the Xamarin libraries, and the platform APIs. The third column refers to any .NET-based Windows Platform regardless of the device:



The boxes in the second row are the actual platform-specific applications. These apps make calls into the common project and also (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. Exactly what version of .NET this is depends on the common code: A PCL has access to its own version of .NET, while an 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 iOS app is built, the Xamarin C# compiler generates C# Intermediate Language (IL) as usual, but it then makes use of the Apple compiler on the Mac to generate native iOS machine code just like the Objective-C compiler. The calls from the app to the iOS APIs are the same as though 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 though 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 an alternative that will simplify your life even more.

Introducing Xamarin.Forms

On May 28, 2014, Xamarin introduced Xamarin.Forms, which allows you to write user-interface code that can be compiled for the iOS, Android, and Windows devices.

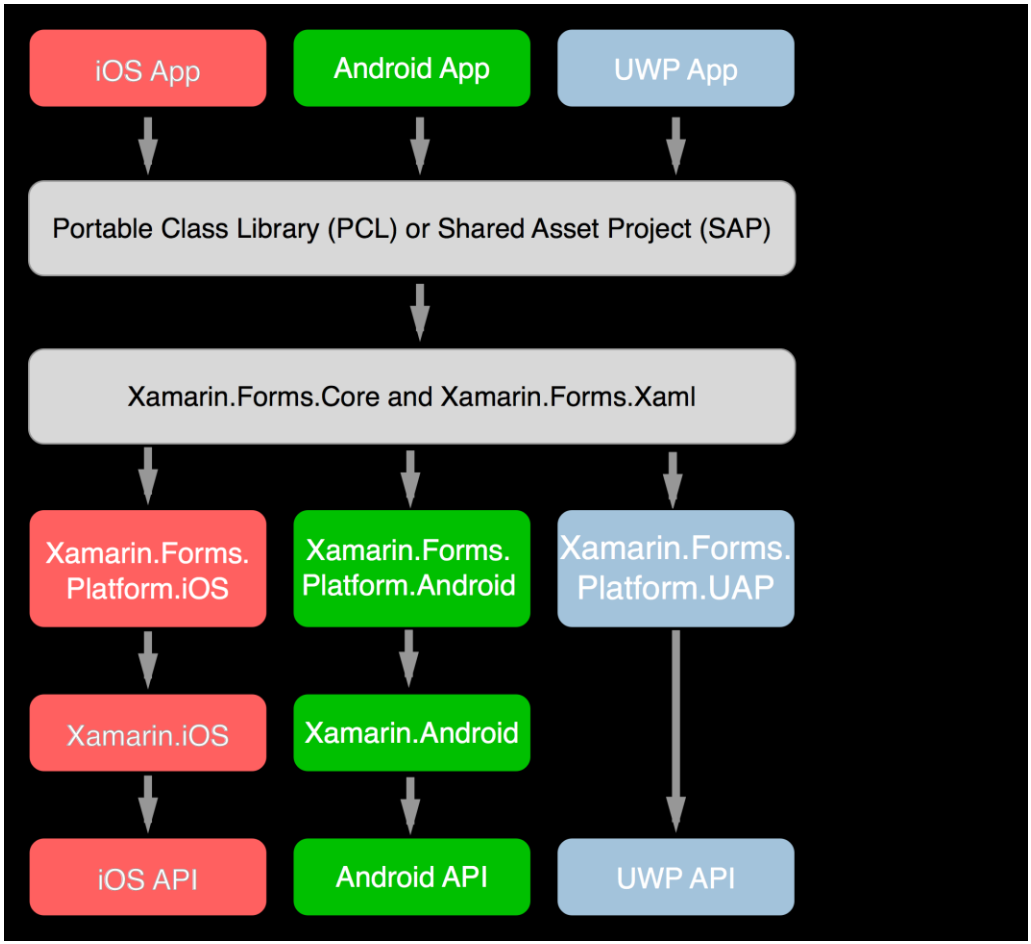
The Xamarin.Forms option

Xamarin.Forms supports five distinct application platforms:

- iOS for programs that run on the iPhone, iPad, and iPod Touch.
- Android for programs that run on Android phones and tablets.
- The Universal Windows Platform (UWP) for applications that runs under Windows 10 or Windows 10 Mobile.
- The Windows Runtime API of Windows 8.1.
- The Windows Runtime API of Windows Phone 8.1.

In this book, "Windows" or "Windows Phone" will generally be used as a generic term to describe all three of the Microsoft platforms.

In the general case, a Xamarin.Forms application in Visual Studio consists of five separate projects for each of these five platforms, with a sixth project containing common code. But the five platform projects in a Xamarin.Forms application are typically quite small—often consisting of just stubs with a little boilerplate startup code. The PCL or SAP contains the bulk of the application, including the user-interface code. The following diagram shows just the iOS, Android, and Universal Windows Platform. The other two Windows platforms are similar to UWP:



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.

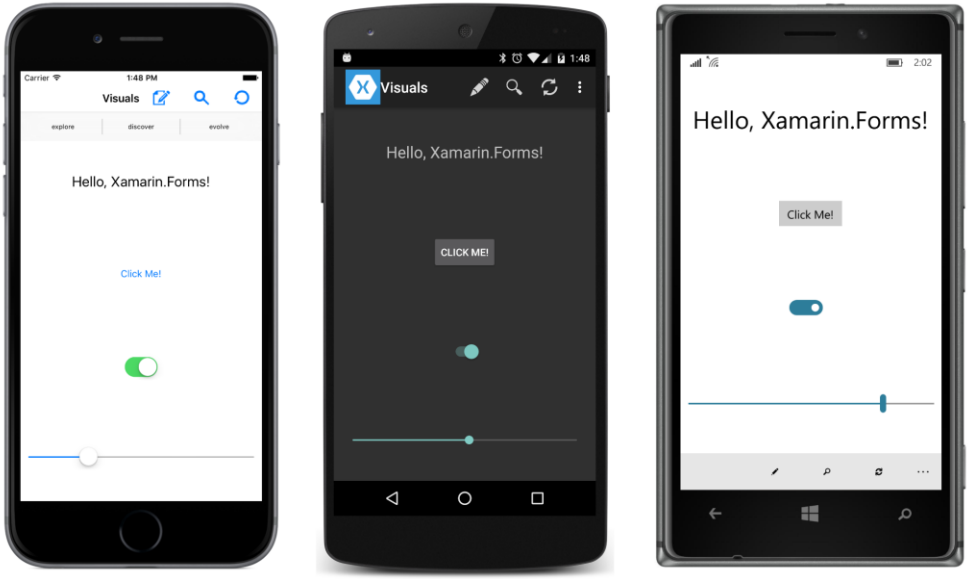
The remainder of the diagram is the same as the one shown earlier.

For example, suppose you need the user-interface object discussed earlier that allows the user to toggle a Boolean value. 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 `ToggleSwitch` 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, an Android phone, and a Windows 10 Mobile device:



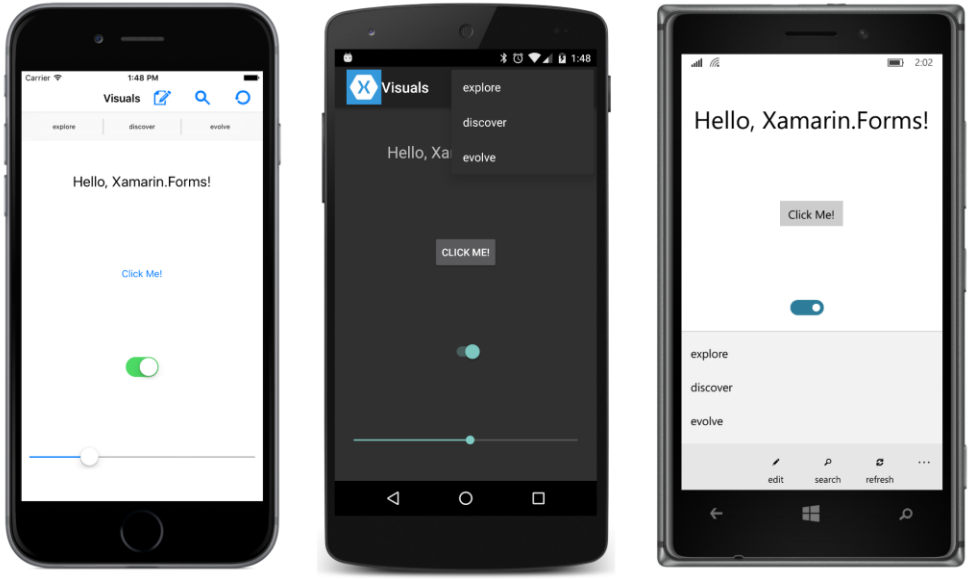
The iPhone screenshot is of an iPhone 6 simulator running iOS 9.2. The Android phone is an LG Nexus 5 running Android version 6. The Windows 10 Mobile device is a Nokia Lumia 935 running a Windows 10 Technical Preview.

You'll encounter triple screenshots like this one throughout this book. They're always in the same order—iPhone, Android, and Windows 10 Mobile—and they're always running the same program.

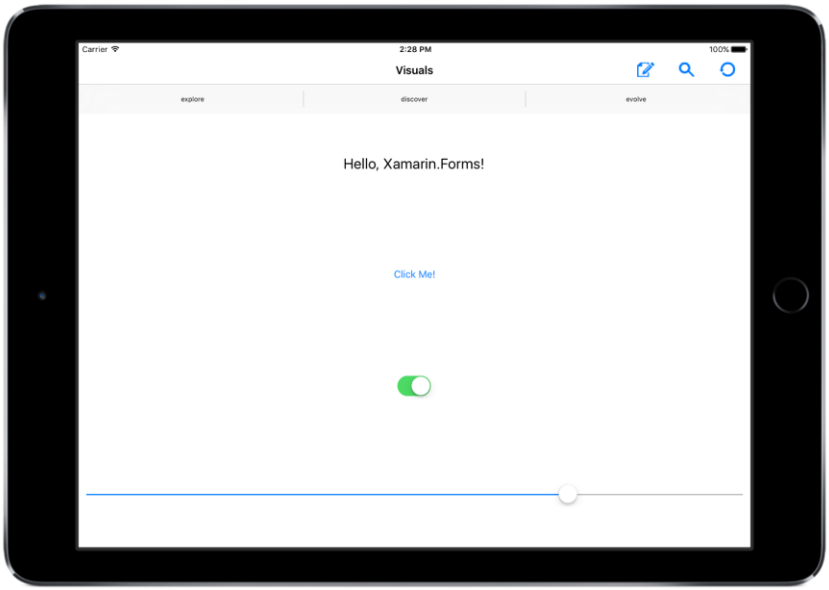
As you can see, the `Button`, `Switch`, and `Slider` all have different appearances on the three phones because they are all rendered with the object specific to each platform.

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 10 Mobile, they're realized as items on the `CommandBar` at the page's bottom.

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

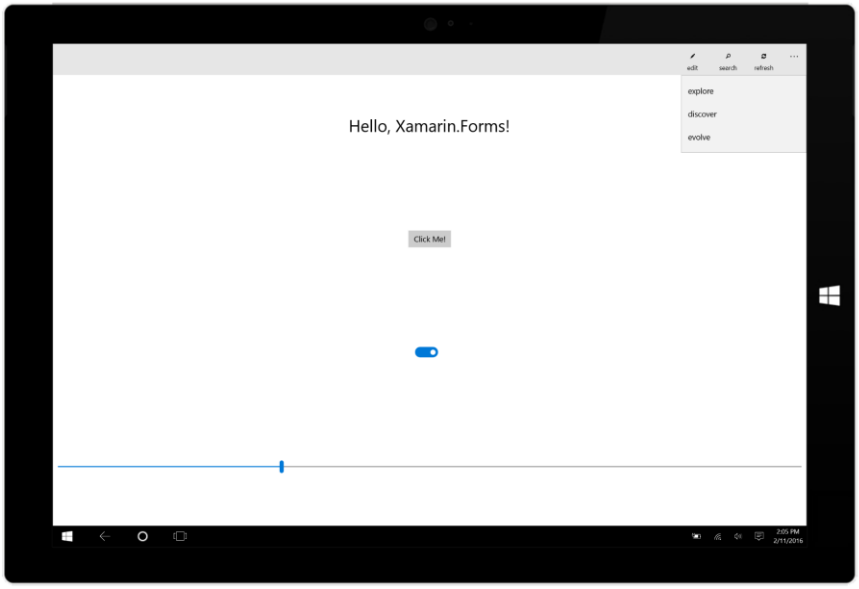


Xamarin.Forms was originally conceived as a platform-independent API for mobile devices. However, Xamarin.Forms is not limited to phones. Here's the same program running on an iPad Air 2 simulator:



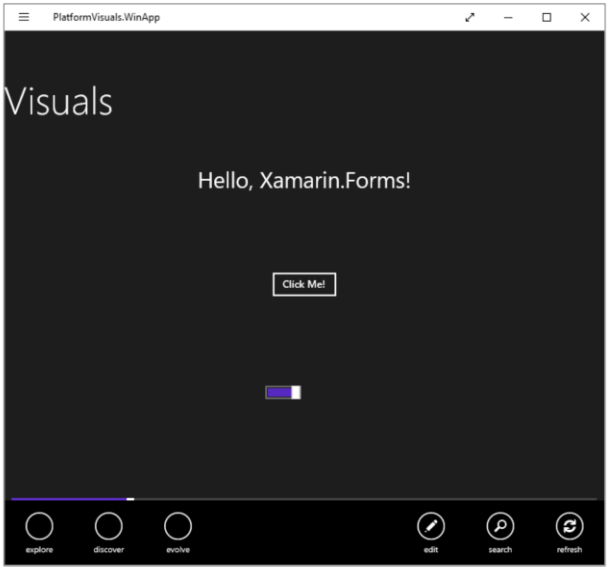
Most of the programs in this book are fairly simple, and hence designed to look their best on a phone screen in portrait mode. But they will also run in landscape mode and on tablets.

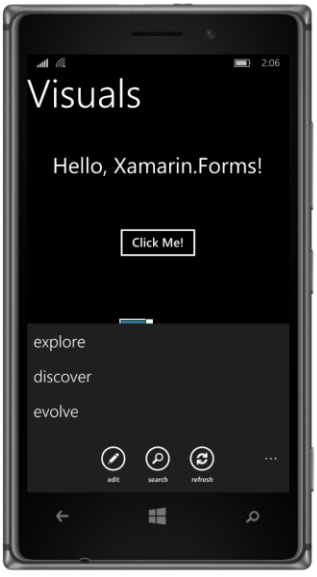
Here's the UWP project on a Microsoft Surface Pro 3 running Windows 10:



Notice the toolbar at the top of the screen. The ellipsis has already been pressed to reveal the three secondary items.

The other two platforms supported by Xamarin.Forms are Windows 8.1 and Windows Phone 8.1. Here's the Windows 8.1 program running in a window on the Windows 10 desktop, and the Windows 8.1 program running on the Windows 10 Mobile device:





The Windows 8.1 screen has been left-clicked with the mouse to reveal the toolbar items at the bottom. On this screen, the secondary items are at the left, but the program neglectfully forgot to assign them icons. On the Windows Phone 8.1 screen, the ellipsis at the bottom has been pressed.

The various implementations of the toolbar reveals that, in one sense, Xamarin.Forms is an API that virtualizes not only the user-interface elements on each platform, but also the user-interface paradigms.

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 how it's been used 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!"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

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

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

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

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="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 Text="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 Text="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 Text="explore" Order="Secondary" />
        <ToolbarItem Text="discover" Order="Secondary" />
        <ToolbarItem Text="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 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.

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

You can create 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 Xamarin.Forms platform libraries, all the corresponding renderers are public classes, and you can derive from them as well.

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 defines its abstractions with a focus on areas of the mobile user interface that are common to the iOS, Android, and Windows Runtime APIs. 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.

You'll probably also want to stick with the native APIs for applications that require vector graphics or complex touch interaction. The current 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 ideal for line-of-business applications.

Even if you begin building an 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 Runtime platforms.

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 Wi-Fi). Visual Studio communicates with the Mac through a Secure Shell (SSH) interface, and uses the Mac to build the application and run the program on a device or simulator.

You can also do Android programming in Xamarin Studio on the Mac or in Visual Studio on the PC.

If you want to target the Windows platforms, you'll need Visual Studio 2015. You can target all the platforms in a single IDE by running Visual Studio 2015 on a PC connected to the Mac via a network. (That's how the sample programs in this book were created.) Another option is to run Visual Studio in a virtual machine on the Mac.

Devices and emulators

You can test your programs on real phones connected to the machines 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.

The iPhone and iPad emulators run on the Mac. However, because Mac desktop machines don't have touchscreens, you'll need to use the mouse or trackpad to simulate touch. The touch gestures on the Mac touchpad 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.

Historically, Android emulators supplied by Google have tended to be slow and cranky, although they are often extremely versatile in emulating a vast array of actual Android devices. Fortunately, Visual Studio now has its own Android emulator that works rather better. It's also very easy to connect

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

The Windows Phone emulators are capable of several different screen resolutions and also tend to run fairly smoothly, albeit consuming lots of memory. If you run the Windows Phone emulator on a touchscreen, you can use touch on the emulator screen. Connecting a real Windows Phone to the PC is fairly easy but requires enabling the phone in the **Settings** section for developing. If you want to unlock more than one phone, you'll need a developer account.

Installation

Before writing applications for Xamarin.Forms, you'll need to install the Xamarin platform on your Mac, PC, or both (if you're using that setup). See the articles on the Xamarin website at:

https://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 normal Windows, Windows Phone, and Windows 10 Mobile projects.

This is important: if you're experiencing a problem using Xamarin.iOS, Xamarin.Android, or Windows, 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:

https://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 Visual Studio, and if everything is installed correctly, 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** (which refers to targeting both iPhone and iPad), and from the template list in the center, select **Blank App (iOS)**.

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

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:

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

If you're using Visual Studio, and if everything is installed correctly, 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)**.

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

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

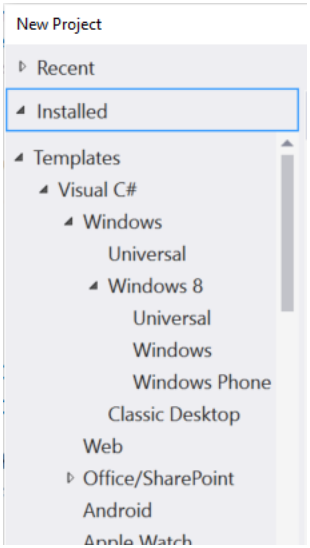
http://forums.xamarin.com/categories/android/

Creating a Windows app

If you're interested in using Xamarin.Forms to target Windows, Windows Phone, or Windows 10 Mobile, you'll need to become familiar with at least the rudiments of using Visual Studio to develop Windows applications:

http://dev.windows.com/

In Visual Studio 2015, if everything is installed correctly, you should be able select **File > New > Project** from the menu, and in the **New Project** dialog, at the left, select **Visual C#** and **Windows**. You'll see a hierarchy under the **Windows** heading something like this:



The first **Universal** heading under **Windows** is for creating a Universal Windows Platform application that can target either Windows 10 or Windows 10 Mobile. Select that, and from the center area select **Blank App (Universal Windows)** to create a UWP app.

The other two project types supported by Xamarin.Forms are under the Windows 8 header. The **Universal** item actually creates two projects—a Windows desktop application and a Windows Phone application with some shared code. For creating just a Windows application, choose **Windows** and then from the center section **Blank App (Windows 8.1)**. For a Windows Phone application, choose **Windows Phone** and **Blank App** This creates a project that targets Windows Phone 8.1.

These are the three project types supported by Xamarin.Forms.

You should be able to build and deploy the skeleton application to the desktop or 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 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.