Chapter 9

Platform-specific API calls

An emergency has arisen. Anyone playing with the **MonkeyTap** game from the previous chapter will quickly come to the conclusion that it desperately needs a very basic enhancement, and it simply cannot be allowed to exist without it.

MonkeyTap needs sound.

It doesn't need very sophisticated sound—just little beeps to accompany the flashes of the four `BoxView` elements. But the Xamarin.Forms API doesn't support sound, so sound is not something we can add to **MonkeyTap** with just a couple of API calls. Supporting sound requires going somewhat beyond Xamarin.Forms to make use of platform-specific sound-generation facilities. Figuring out how to make sounds in iOS, Android, and Windows Phone is hard enough. But how does a Xamarin.Forms program then make calls into the individual platforms?

Before tackling the complexities of sound, let's examine the different approaches to making platform-specific API calls with a much simpler example. The first three short programs shown in this chapter are all functionally identical: They all display two tiny items of information supplied by the underlying platform's operating system that reveal the model of the device running the program and the operating system version.

Preprocessing in the Shared Asset Project

As you learned in Chapter 2, "Anatomy of an app," you can use either a Shared Asset Project (SAP) or a Portable Class Library (PCL) for the code that is common to all three platforms. An SAP contains code files that are shared among the platform projects, while a PCL encloses the common code in a library that is accessible only through public types.

Accessing platform APIs from a Shared Asset Project is a little more straightforward than from a Portable Class Library because it involves more traditional programming tools, so let's try that approach first. You can create a Xamarin.Forms solution with an SAP using the process described in Chapter 2. You can then add a XAML-based `ContentPage` class to the SAP the same way you add one to a PCL.

Here's the XAML file for a project that displays platform information, named **PlatInfoSap1**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PlatInfoSap1.PlatInfoSap1Page">

    <StackLayout Padding="20">
```

```
        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Device Model:" />

            <ContentView Padding="50, 0, 0, 0">
                <Label x:Name="modelLabel"
                       FontSize="Large"
                       FontAttributes="Bold" />
            </ContentView>
        </StackLayout>

        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Operating System Version:" />

            <ContentView Padding="50, 0, 0, 0">
                <Label x:Name="versionLabel"
                       FontSize="Large"
                       FontAttributes="Bold" />
            </ContentView>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The code-behind file must set the `Text` properties for `modelLabel` and `versionLabel`.

Code files in a Shared Asset Project are extensions of the code in the individual platforms. This means that code in the SAP can make use of the C# preprocessor directives `#if`, `#elif`, `#else`, and `#endif` with conditional-compilation symbols defined for the three platforms, as demonstrated in Chapters 2 and 4. These symbols are:

- `__IOS__` for iOS
- `__ANDROID__` for Android
- `WINDOWS_UWP` for the Universal Windows Platform
- `WINDOWS_APP` for Windows 8.1
- `WINDOWS_PHONE_APP` for Windows Phone 8.1

The APIs involved in obtaining the model and version information are, of course, different for the three platforms:

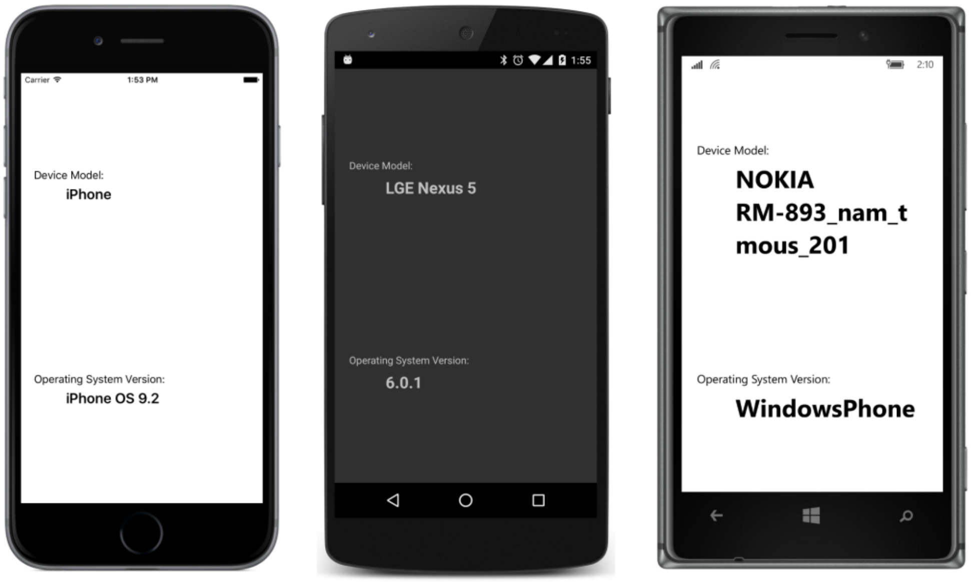
- For iOS, use the `UIDevice` class in the `UIKit` namespace.
- For Android, use various properties of the `Build` class in the `Android.OS` namespace.
- For the Windows platforms, use the `EasClientDeviceInformation` class in the `Windows.Security.ExchangeActiveSyncProvisioning` namespace.

Here's the PlatInfoSap1.xaml.cs code-behind file showing how `modelLabel` and `versionLabel` are set based on the conditional-compilation symbols:

```
using System;
using Xamarin.Forms;

#if __IOS__
using UIKit;

#elif __ANDROID__
using Android.OS;

#elif WINDOWS_APP || WINDOWS_PHONE_APP || WINDOWS_UWP
using Windows.Security.ExchangeActiveSyncProvisioning;

#endif

namespace PlatInfoSap1
{
    public partial class PlatInfoSap1Page : ContentPage
    {
        public PlatInfoSap1Page ()
        {
            InitializeComponent ();

#if __IOS__

            UIDevice device = new UIDevice();
            modelLabel.Text = device.Model.ToString();
            versionLabel.Text = String.Format("{0} {1}", device.SystemName,
                                                         device.SystemVersion);

#elif __ANDROID__

            modelLabel.Text = String.Format("{0} {1}", Build.Manufacturer,
                                                       Build.Model);
            versionLabel.Text = Build.VERSION.Release.ToString();

#elif WINDOWS_APP || WINDOWS_PHONE_APP || WINDOWS_UWP

            EasClientDeviceInformation devInfo = new EasClientDeviceInformation();
            modelLabel.Text = String.Format("{0} {1}", devInfo.SystemManufacturer,
                                                       devInfo.SystemProductName);
            versionLabel.Text = devInfo.OperatingSystem;

#endif

        }
    }
}
```

Notice that these preprocessor directives are used to select different `using` directives as well as to make calls to platform-specific APIs. In a program as simple as this, you could simply include the namespaces with the class names, but for longer blocks of code, you'll probably want those `using` directives.

And of course it works:


Device Model:
iPhone
Operating System Version:
iPhone OS 9.2

Device Model:
LGE Nexus 5
Operating System Version:
6.0.1

Device Model:
NOKIA RM-893_nam_tmous_201
Operating System Version:
WindowsPhone

The advantage of this approach is that you have all the code for the three platforms in one place. But the preprocessor directives in the code listing are—let's face it—rather ugly, and they harken back to a much earlier era in programming. Using preprocessor directives might not seem so bad for short and less frequent calls such as this example, but in a larger program you'll need to juggle blocks of platform-specific code and shared code, and the multitude of preprocessor directives can easily become confusing. Preprocessor directives should be used for little fixes and generally not as structural elements in the application.

Let's try another approach.

Parallel classes and the Shared Asset Project

Although the Shared Asset Project is an extension of the platform projects, the relationship goes both ways: just as a platform project can make calls into code in a Shared Asset Project, the SAP can make calls into the individual platform projects.

This means that we can restrict the platform-specific API calls to classes in the individual platform projects. If the names and namespaces of these classes in the platform projects are the same, then code in the SAP can access these classes in a transparent, platform-independent manner.

In the **PlatInfoSap2** solution, each of the five platform projects has a class named `PlatformInfo` that contains two methods that return `string` objects, named `GetModel` and `GetVersion`. Here's the version of this class in the iOS project:

```
using System;
using UIKit;

namespace PlatInfoSap2
{
    public class PlatformInfo
    {
        UIDevice device = new UIDevice();

        public string GetModel()
        {
            return device.Model.ToString();
        }

        public string GetVersion()
        {
            return String.Format("{0} {1}", device.SystemName,
                                            device.SystemVersion);
        }
    }
}
```

Notice the namespace name. Although the other classes in this iOS project use the `PlatInfo-Sap2.iOS` namespace, the namespace for this class is just `PlatInfoSap2`. This allows the SAP to access this class directly without any platform specifics.

Here's the parallel class in the Android project. Same namespace, same class name, and same method names, but different implementations of these methods using Android API calls:

```
using System;
using Android.OS;

namespace PlatInfoSap2
{
    public class PlatformInfo
    {
        public string GetModel()
        {
            return String.Format("{0} {1}", Build.Manufacturer,
                                            Build.Model);
        }

        public string GetVersion()
        {
            return Build.VERSION.Release.ToString();
        }
    }
}
```

And here's the class that exists in three identical copies in the three Windows and Windows Phone projects:

```
using System;
using Windows.Security.ExchangeActiveSyncProvisioning;
```

```
namespace PlatInfoSap2
{
    public class PlatformInfo
    {
        EasClientDeviceInformation devInfo = new EasClientDeviceInformation();

        public string GetModel()
        {
            return String.Format("{0} {1}", devInfo.SystemManufacturer,
                                            devInfo.SystemProductName);
        }

        public string GetVersion()
        {
            return devInfo.OperatingSystem;
        }
    }
}
```

The XAML file in the **PlatInfoSap2** project is basically the same as the one in **PlatInfoSap1** project. The code-behind file is considerably simpler:

```
using System;
using Xamarin.Forms;

namespace PlatInfoSap2
{
    public partial class PlatInfoSap2Page : ContentPage
    {
        public PlatInfoSap2Page ()
        {
            InitializeComponent ();

            PlatformInfo platformInfo = new PlatformInfo();
            modelLabel.Text = platformInfo.GetModel();
            versionLabel.Text = platformInfo.GetVersion();
        }
    }
}
```

The particular version of `PlatformInfo` that is referenced by the class is the one in the compiled project. It's almost as if we've defined a little extension to Xamarin.Forms that resides in the individual platform projects.

DependencyService and the Portable Class Library

Can the technique illustrated in the **PlatInfoSap2** program be implemented in a solution with a Portable Class Library? At first, it doesn't seem possible. Although application projects make calls to libraries all the time, libraries generally can't make calls to applications except in the context of events or callback functions. The PCL is bundled with a device-independent version of .NET and closed up tight—capable only of executing code within itself or other PCLs it might reference.

But wait: When a Xamarin.Forms application is running, it can use .NET reflection to get access to its own assembly and any other assemblies in the program. This means that code in the PCL can use reflection to access classes that exist in the platform assembly from which the PCL is referenced. Those classes must be defined as public, of course, but that's just about the only requirement.

Before you start writing code that exploits this technique, you should know that this solution already exists in the form of a Xamarin.Forms class named `DependencyService`. This class uses .NET reflection to search through all the other assemblies in the application—including the particular platform assembly itself—and provide access to platform-specific code.

The use of `DependencyService` is illustrated in the **DisplayPlatformInfo** solution, which uses a Portable Class Library for the shared code. You begin the process of using `DependencyService` by defining an interface type in the PCL project that declares the signatures of the methods you want to implement in the platform projects. Here's `IPlatformInfo`:

```
namespace DisplayPlatformInfo
{
    public interface IPlatformInfo
    {
        string GetModel();

        string GetVersion();
    }
}
```

You've seen those two methods before. They're the same two methods implemented in the `PlatformInfo` classes in the platform projects in **PlatInfoSap2**.

In a manner very similar to **PlatInfoSap2**, all three platform projects in **DisplayPlatformInfo** must now have a class that implements the `IPlatformInfo` interface. Here's the class in the iOS project, named `PlatformInfo`:

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

[assembly: Dependency(typeof(DisplayPlatformInfo.iOS.PlatformInfo))]

namespace DisplayPlatformInfo.iOS
{
    public class PlatformInfo : IPlatformInfo
    {
        UIDevice device = new UIDevice();

        public string GetModel()
        {
            return device.Model.ToString();
        }

        public string GetVersion()
        {
            return String.Format("{0} {1}", device.SystemName,
                                                device.SystemVersion);
        }
    }
}
```

This class is not referenced directly from the PCL, so the namespace name can be anything you want. Here it's set to the same namespace as the other code in the iOS project. The class name can also be anything you want. Whatever you name it, however, the class must explicitly implement the `IPlatformInfo` interface defined in the PCL:

```
public class PlatformInfo : IPlatformInfo
```

Furthermore, this class must be referenced in a special attribute outside the namespace block. You'll see it near the top of the file following the `using` directives:

```
[assembly: Dependency(typeof(DisplayPlatformInfo.iOS.PlatformInfo))]
```

The `DependencyAttribute` class that defines this `Dependency` attribute is part of Xamarin.Forms and used specifically in connection with `DependencyService`. The argument is a `Type` object of a class in the platform project that is available for access by the PCL. In this case, it's this `PlatformInfo` class. This attribute is attached to the platform assembly itself, so code executing in the PCL doesn't have to search all over the library to find it.

Here's the Android version of `PlatformInfo`:

```
using System;
using Android.OS;
using Xamarin.Forms;

[assembly: Dependency(typeof(DisplayPlatformInfo.Droid.PlatformInfo))]

namespace DisplayPlatformInfo.Droid
{
    public class PlatformInfo : IPlatformInfo
    {
        public string GetModel()
        {
            return String.Format("{0} {1}", Build.Manufacturer,
                                                Build.Model);
        }

        public string GetVersion()
        {
            return Build.VERSION.Release.ToString();
        }
    }
}
```

And here's the one for the UWP project:

```
using System;
using Windows.Security.ExchangeActiveSyncProvisioning;
using Xamarin.Forms;

[assembly: Dependency(typeof(DisplayPlatformInfo.UWP.PlatformInfo))]

namespace DisplayPlatformInfo.UWP
{
    public class PlatformInfo : IPlatformInfo
    {
        EasClientDeviceInformation devInfo = new EasClientDeviceInformation();

        public string GetModel()
        {
            return String.Format("{0} {1}", devInfo.SystemManufacturer,
                                            devInfo.SystemProductName);
        }

        public string GetVersion()
        {
            return devInfo.OperatingSystem;
        }
    }
}
```

The Windows 8.1 and Windows Phone 8.1 projects have similar files that differ only by the namespace.

Code in the PCL can then get access to the particular platform's implementation of `IPlatform-Info` by using the `DependencyService` class. This is a static class with three public methods, the most important of which is named `Get`. `Get` is a generic method whose argument is the interface you've de-fined, in this case `IPlatformInfo`.

```
IPlatformInfo platformInfo = DependencyService.Get<IPlatformInfo>();
```

The `Get` method returns an instance of the platform-specific class that implements the `IPlatform-Info` interface. You can then use this object to make platform-specific calls. This is demonstrated in the code-behind file for the **DisplayPlatformInfo** project:

```
namespace DisplayPlatformInfo
{
    public partial class DisplayPlatformInfoPage : ContentPage
    {
        public DisplayPlatformInfoPage()
        {
            InitializeComponent();

            IPlatformInfo platformInfo = DependencyService.Get<IPlatformInfo>();
            modelLabel.Text = platformInfo.GetModel();
            versionLabel.Text = platformInfo.GetVersion();
        }
    }
}
```

`DependencyService` caches the instances of the objects that it obtains through the `Get` method. This speeds up subsequent uses of `Get` and also allows the platform implementations of the interface to maintain state: any fields and properties in the platform implementations will be preserved across multiple `Get` calls. These classes can also include events or implement callback methods.

`DependencyService` requires just a little more overhead than the approach shown in the **PlatInfoSap2** project and is somewhat more structured because the individual platform classes implement an interface defined in shared code.

`DependencyService` is not the only way to implement platform-specific calls in a PCL. Adventurous developers might want to use dependency-injection techniques to configure the PCL to make calls into the platform projects. But `DependencyService` is very easy to use, and it eliminates most reasons to use a Shared Asset Project in a Xamarin.Forms application.

Platform-specific sound generation

Now for the real objective of this chapter: to give sound to **MonkeyTap**. All three platforms support APIs that allow a program to dynamically generate and play audio waveforms. This is the approach taken by the **MonkeyTapWithSound** program.

Commercial music files are often compressed in formats such as MP3. But when a program is algorithmically generating waveforms, an uncompressed format is much more convenient. The most basic technique—which is supported by all three platforms—is called pulse code modulation or PCM. Despite the fancy name, it's quite simple, and it's the technique used for storing sound on music CDs.

A PCM waveform is described by a series of samples at a constant rate, known as the sampling rate. Music CDs use a standard rate of 44,100 samples per second. Audio files generated by computer programs often use a sampling rate of half that (22,050) or one-quarter (11,025) if high audio quality is not required. The highest frequency that can be recorded and reproduced is one-half the sampling rate.

Each sample is a fixed size that defines the amplitude of the waveform at that point in time. The samples on a music CD are signed 16-bit values. Samples of 8 bits are common when sound quality doesn't matter as much. Some environments support floating-point values. Multiple samples can accommodate stereo or any number of channels. For simple sound effects on mobile devices, monaural sound is often fine.

The sound generation algorithm in **MonkeyTapWithSound** is hard-coded for 16-bit monaural samples, but the sampling rate is specified by a constant and can easily be changed.

Now that you know how `DependencyService` works, let's examine the code added to **Monkey-**

Tap to turn it into **MonkeyTapWithSound**, and let's look at it from the top down. To avoid reproducing a lot of code, the new project contains links to the MonkeyTap.xaml and MonkeyTap.xaml.cs files in the **MonkeyTap** project.

In Visual Studio, you can add items to projects as links to existing files by selecting **Add > Existing Item** from the project menu. Then use the **Add Existing Item** dialog to navigate to the file. Choose **Add as Link** from the drop-down on the **Add** button.

In Xamarin Studio, select **Add > Add Files** from the project's tool menu. After opening the file or files, an **Add File to Folder** alert box pops up. Choose **Add a link to the file**.

However, after taking these steps in Visual Studio, it was also necessary to manually edit the MonkeyTapWithSound.csproj file to change the MonkeyTapPage.xaml file to an **EmbeddedResource** and the **Generator** to **MSBuild:UpdateDesignTimeXaml**. Also, a **DependentUpon** tag was added to the MonkeyTapPage.xaml.cs file to reference the MonkeyTapPage.xaml file. This causes the code-behind file to be indented under the XAML file in the file list.

The `MonkeyTapWithSoundPage` class then derives from the `MonkeyTapPage` class. Although the `MonkeyTapPage` class is defined by a XAML file and a code-behind file, `MonkeyTapWithSoundPage` is code only. When a class is derived in this way, event handlers in the original code-behind file for events in the XAML file must be defined as `protected`, and this is the case.

The `MonkeyTap` class also defined a `flashDuration` constant as `protected`, and two methods were defined as `protected` and `virtual`. The `MonkeyTapWithSoundPage` overrides these two methods to call a static method named `SoundPlayer.PlaySound`:

```
namespace MonkeyTapWithSound
{
    class MonkeyTapWithSoundPage : MonkeyTap.MonkeyTapPage
    {
        const int errorDuration = 500;

        // Diminished 7th in 1st inversion: C, Eb, F#, A
        double[] frequencies = { 523.25, 622.25, 739.99, 880 };

        protected override void BlinkBoxView(int index)
        {
            SoundPlayer.PlaySound(frequencies[index], flashDuration);
            base.BlinkBoxView(index);
        }

        protected override void EndGame()
        {
            SoundPlayer.PlaySound(65.4, errorDuration);
            base.EndGame();
        }
    }
}
```

The `SoundPlayer.PlaySound` method accepts a frequency and a duration in milliseconds. Everything else—the volume, the harmonic makeup of the sound, and how the sound is generated—is the responsibility of the `PlaySound` method. However, this code makes an implicit assumption that `SoundPlayer.PlaySound` returns immediately and does not wait for the sound to complete playing. Fortunately, all three platforms support sound-generation APIs that behave in this way.

The `SoundPlayer` class with the `PlaySound` static method is part of the **MonkeyTapWithSound** PCL project. The responsibility of this method is to define an array of the PCM data for the sound. The size of this array is based on the sampling rate and the duration. The `for` loop calculates samples that define a triangle wave of the requested frequency:

```
namespace MonkeyTapWithSound
{
    class SoundPlayer
    {
        const int samplingRate = 22050;

        // Hard-coded for monaural, 16-bit-per-sample PCM
        public static void PlaySound(double frequency = 440, int duration = 250)
        {
            short[] shortBuffer = new short[samplingRate * duration / 1000];
            double angleIncrement = frequency / samplingRate;
            double angle = 0;   // normalized 0 to 1

            for (int i = 0; i < shortBuffer.Length; i++)
            {
                // Define triangle wave
                double sample;

                // 0 to 1
                if (angle < 0.25)
                    sample = 4 * angle;

                // 1 to -1
                else if (angle < 0.75)
                    sample = 4 * (0.5 - angle);

                // -1 to 0
                else
                    sample = 4 * (angle - 1);

                shortBuffer[i] = (short)(32767 * sample);
                angle += angleIncrement;

                while (angle > 1)
                    angle -= 1;
            }

            byte[] byteBuffer = new byte[2 * shortBuffer.Length];
            Buffer.BlockCopy(shortBuffer, 0, byteBuffer, 0, byteBuffer.Length);

            DependencyService.Get<IPlatformSoundPlayer>().PlaySound(samplingRate, byteBuffer);
        }
    }
}
```

Although the samples are 16-bit integers, two of the platforms want the data in the form of an array of bytes, so a conversion occurs near the end with `Buffer.BlockCopy`. The last line of the method uses `DependencyService` to pass this byte array with the sampling rate to the individual platforms.

The `DependencyService.Get` method references the `IPlatformSoundPlayer` interface that defines the signature of the `PlaySound` method:

```
namespace MonkeyTapWithSound
{
    public interface IPlatformSoundPlayer
    {
        void PlaySound(int samplingRate, byte[] pcmData);
    }
}
```

Now comes the hard part: writing this `PlaySound` method for the three platforms!

The iOS version uses `AVAudioPlayer`, which requires data that includes the header used in Waveform Audio File Format (.wav) files. The code here assembles that data in a `MemoryBuffer` and then converts that to an `NSData` object:

```
using System;
using System.IO;
using System.Text;
using Xamarin.Forms;
using AVFoundation;
using Foundation;

[assembly: Dependency(typeof(MonkeyTapWithSound.iOS.PlatformSoundPlayer))]

namespace MonkeyTapWithSound.iOS
{
    public class PlatformSoundPlayer : IPlatformSoundPlayer
    {
        const int numChannels = 1;
        const int bitsPerSample = 16;

        public void PlaySound(int samplingRate, byte[] pcmData)
        {
            int numSamples = pcmData.Length / (bitsPerSample / 8);

            MemoryStream memoryStream = new MemoryStream();
            BinaryWriter writer = new BinaryWriter(memoryStream, Encoding.ASCII);

            // Construct WAVE header.
            writer.Write(new char[] { 'R', 'I', 'F', 'F' });
            writer.Write(36 + sizeof(short) * numSamples);
            writer.Write(new char[] { 'W', 'A', 'V', 'E' });
            writer.Write(new char[] { 'f', 'm', 't', ' ' });                // format chunk
            writer.Write(16);                                              // PCM chunk size
            writer.Write((short)1);                                        // PCM format flag
            writer.Write((short)numChannels);
            writer.Write(samplingRate);
            writer.Write(samplingRate * numChannels * bitsPerSample / 8);  // byte rate
            writer.Write((short)(numChannels * bitsPerSample / 8));        // block align
            writer.Write((short)bitsPerSample);
            writer.Write(new char[] { 'd', 'a', 't', 'a' });               // data chunk
            writer.Write(numSamples * numChannels * bitsPerSample / 8);

            // Write data as well.
            writer.Write(pcmData, 0, pcmData.Length);

            memoryStream.Seek(0, SeekOrigin.Begin);
            NSData data = NSData.FromStream(memoryStream);
            AVAudioPlayer audioPlayer = AVAudioPlayer.FromData(data);
            audioPlayer.Play();
        }
    }
}
```

Notice the two essentials: `PlatformSoundPlayer` implements the `IPlatformSoundPlayer` interface, and the class is flagged with the `Dependency` attribute.

The Android version uses the `AudioTrack` class, and that turns out to be a little easier. However, `AudioTrack` objects can't overlap, so it's necessary to save the previous object and stop it playing before starting the next one:

```
using System;
using Android.Media;
using Xamarin.Forms;

[assembly: Dependency(typeof(MonkeyTapWithSound.Droid.PlatformSoundPlayer))]

namespace MonkeyTapWithSound.Droid
{
    public class PlatformSoundPlayer : IPlatformSoundPlayer
    {
        AudioTrack previousAudioTrack;

        public void PlaySound(int samplingRate, byte[] pcmData)
        {
            if (previousAudioTrack != null)
            {
                previousAudioTrack.Stop();
                previousAudioTrack.Release();
            }

            AudioTrack audioTrack = new AudioTrack(Stream.Music,
                                                   samplingRate,
                                                   ChannelOut.Mono,
                                                   Android.Media.Encoding.Pcm16bit,
                                                   pcmData.Length * sizeof(short),
                                                   AudioTrackMode.Static);

            audioTrack.Write(pcmData, 0, pcmData.Length);
            audioTrack.Play();

            previousAudioTrack = audioTrack;
        }
    }
}
```

The three Windows and Windows Phone platforms can use `MediaStreamSource`. To avoid a lot of repetitive code, the **MonkeyTapWithSound** solution contains an additional SAP project named **WinRuntimeShared** consisting solely of a class that all three platforms can use:

```
using System;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Media.Core;
using Windows.Media.MediaProperties;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Controls;

namespace MonkeyTapWithSound.WinRuntimeShared
{
    public class SharedSoundPlayer
    {
        MediaElement mediaElement = new MediaElement();
        TimeSpan duration;

        public void PlaySound(int samplingRate, byte[] pcmData)
        {
            AudioEncodingProperties audioProps =
                AudioEncodingProperties.CreatePcm((uint)samplingRate, 1, 16);
            AudioStreamDescriptor audioDesc = new AudioStreamDescriptor(audioProps);
            MediaStreamSource mss = new MediaStreamSource(audioDesc);

            bool samplePlayed = false;
            mss.SampleRequested += (sender, args) =>
            {
                if (samplePlayed)
                    return;

                IBuffer ibuffer = pcmData.AsBuffer();
                MediaStreamSample sample =
                    MediaStreamSample.CreateFromBuffer(ibuffer, TimeSpan.Zero);
                sample.Duration = TimeSpan.FromSeconds(pcmData.Length / 2.0 / samplingRate);
                args.Request.Sample = sample;
                samplePlayed = true;
            };

            mediaElement.SetMediaStreamSource(mss);
        }
    }
}
```

This SAP project is referenced by the three Windows and Windows Phone projects, each of which contains an identical (except for the namespace) `PlatformSoundPlayer` class:

```
using System;
using Xamarin.Forms;

[assembly: Dependency(typeof(MonkeyTapWithSound.UWP.PlatformSoundPlayer))]

namespace MonkeyTapWithSound.UWP
{
    public class PlatformSoundPlayer : IPlatformSoundPlayer
    {
        WinRuntimeShared.SharedSoundPlayer sharedSoundPlayer;

        public void PlaySound(int samplingRate, byte[] pcmData)
        {
            if (sharedSoundPlayer == null)
            {
                sharedSoundPlayer = new WinRuntimeShared.SharedSoundPlayer();
            }

            sharedSoundPlayer.PlaySound(samplingRate, pcmData);
        }
    }
}
```

The use of `DependencyService` to perform platform-specific chores is very powerful, but this approach falls short when it comes to user-interface elements. If you need to expand the arsenal of views that adorn the pages of your Xamarin.Forms applications, that job involves creating platform-specific renderers, a process discussed in the final chapter of this book.

Chapter 10
XAML markup extensions

In code, you can set a property in a variety of different ways from a variety of different sources:

```
triangle.Angle1 = 45;
triangle.Angle1 = 180 * radians / Math.PI;
triangle.Angle1 = angles[i];
triangle.Angle1 = animator.GetCurrentAngle();
```

If this `Angle1` property is a `double`, all that's required is that the source be a `double` or otherwise provide a numeric value that is convertible to a `double`.

In markup, however, a property of type `double` usually can be set only from a string that qualifies as a valid argument to `Double.Parse`. The only exception you've seen so far is when the target property is flagged with a `TypeConverter` attribute, such as the `FontSize` property.

It might be desirable if XAML were more flexible—if you could set a property from sources other than explicit text strings. For example, suppose you want to define another way to set a property of type `Color`, perhaps using the `Hue`, `Saturation`, and `Luminosity` values but without the hassle of the `x:FactoryMethod` element. Just offhand, it doesn't seem possible. The XAML parser expects that any value set to an attribute of type `Color` is a string acceptable to the `ColorTypeConverter` class.

The purpose of XAML *markup extensions* is to get around this apparent restriction. Rest assured that XAML markup extensions are *not* extensions to XML. XAML is always legal XML. XAML markup extensions are extensions only in the sense that they extend the possibilities of attribute settings in markup. A markup extension essentially *provides* a value of a particular type without necessarily being a text representation *of* a value.

The code infrastructure

Strictly speaking, a XAML markup extension is a class that implements `IMarkupExtension`, which is a public interface defined in the regular **Xamarin.Forms.Core** assembly but with the namespace `Xamarin.Forms.Xaml`:

```
public interface IMarkupExtension
{
    object ProvideValue(IServiceProvider serviceProvider);
}
```

As the name suggests, `ProvideValue` is the method that provides a value to a XAML attribute. `IServiceProvider` is part of the base class libraries of .NET and defined in the `System` namespace:

```
public interface IServiceProvider
{
    object GetService(Type type);
}
```

Obviously, this information doesn't provide much of a hint on writing custom markup extensions, and in truth, they can be tricky. (You'll see an example shortly and other examples later in this book.) Fortunately, Xamarin.Forms provides several valuable markup extensions for you. These fall into three categories:

- Markup extensions that are part of the XAML 2009 specification. These appear in XAML files with the customary `x` prefix and are:
    - `x:Static`
    - `x:Reference`
    - `x:Type`
    - `x:Null`
    - `x:Array`

  These are implemented in classes that consist of the name of the markup extension with the word `Extension` appended—for example, the `StaticExtension` and `ReferenceExtension` classes. These classes are defined in the **Xamarin.Forms.Xaml** assembly.

- The following markup extensions originated in the Windows Presentation Foundation (WPF) and, with the exception of `DynamicResource`, are supported by Microsoft's other implementations of XAML, including Silverlight, Windows Phone 7 and 8, and Windows 8 and 10:
    - `StaticResource`
    - `DynamicResource`
    - `Binding`

  These are implemented in the public `StaticResourceExtension`, `DynamicResourceExtension`, and `BindingExtension` classes.

- There is only one markup extension that is unique to Xamarin.Forms: the `ConstraintExpression` class used in connection with `RelativeLayout`.

Although it's possible to play around with public markup-extension classes in code, they really only make sense in XAML.

Accessing static members

One of the simplest and most useful implementations of `IMarkupExtension` is encapsulated in the `StaticExtension` class. This is part of the original XAML specification, so it customarily appears in XAML with an `x` prefix. `StaticExtension` defines a single property named `Member` of type `string` that you set to a class and member name of a public constant, static property, static field, or enumeration member.

Let's see how this works. Here's a `Label` with six properties set as they would normally appear in XAML.

```
<Label Text="Just some text"
       BackgroundColor="Accent"
       TextColor="Black"
       FontAttributes="Italic"
       VerticalOptions="Center"
       HorizontalTextAlignment="Center" />
```

Five of these attributes are set to text strings that eventually reference various static properties, fields, and enumeration members, but the conversion of those text strings occurs through type converters and the standard XAML parsing of enumeration types.

If you want to be more explicit in setting these attributes to those various static properties, fields, and enumeration members, you can use `x:StaticExtension` within property element tags:

```
<Label Text="Just some text">
    <Label.BackgroundColor>
        <x:StaticExtension Member="Color.Accent" />
    </Label.BackgroundColor>

    <Label.TextColor>
        <x:StaticExtension Member="Color.Black" />
    </Label.TextColor>

    <Label.FontAttributes>
        <x:StaticExtension Member="FontAttributes.Italic" />
    </Label.FontAttributes>

    <Label.VerticalOptions>
        <x:StaticExtension Member="LayoutOptions.Center" />
    </Label.VerticalOptions>

    <Label.HorizontalTextAlignment>
        <x:StaticExtension Member="TextAlignment.Center" />
    </Label.HorizontalTextAlignment>
</Label>
```

`Color.Accent` is a static property. `Color.Black` and `LayoutOptions.Center` are static fields. `FontAttributes.Italic` and `TextAlignment.Center` are enumeration members.

Considering the ease with which these attributes are set with text strings, the approach using StaticExtension initially seems ridiculous, but notice that it's a general-purpose mechanism. You can use *any* static property, field, or enumeration member in the StaticExtension tag if its type matches the type of the target property.

By convention, classes that implement IMarkupExtension incorporate the word Extension in their names, but you can leave that out in XAML, which is why this markup extension is usually called x:Static rather than x:StaticExtension. The following markup is marginally shorter than the previous block:

```
<Label Text="Just some text">
    <Label.BackgroundColor>
        <x:Static Member="Color.Accent" />
    </Label.BackgroundColor>

    <Label.TextColor>
        <x:Static Member="Color.Black" />
    </Label.TextColor>

    <Label.FontAttributes>
        <x:Static Member="FontAttributes.Italic" />
    </Label.FontAttributes>

    <Label.VerticalOptions>
        <x:Static Member="LayoutOptions.Center" />
    </Label.VerticalOptions>

    <Label.HorizontalTextAlignment>
        <x:Static Member="TextAlignment.Center" />
    </Label.HorizontalTextAlignment>
</Label>
```

And now for the really major markup reduction—a change in syntax that causes the property-element tags to disappear and the footprint to shrink considerably. XAML markup extensions almost always appear with the markup extension name and the arguments within a pair of curly braces:

```
<Label Text="Just some text"
       BackgroundColor="{x:Static Member=Color.Accent}"
       TextColor="{x:Static Member=Color.Black}"
       FontAttributes="{x:Static Member=FontAttributes.Italic}"
       VerticalOptions="{x:Static Member=LayoutOptions.Center}"
       HorizontalTextAlignment="{x:Static Member=TextAlignment.Center}" />
```

This syntax with the curly braces is so ubiquitously used in connection with XAML markup extensions that many developers consider markup extensions to be synonymous with the curly-brace syntax. And that's nearly true: while curly braces always signal the presence of a XAML markup extension, in many cases a markup extension can appear in XAML without the curly braces (as demonstrated earlier) and it's sometimes convenient to use them in that way.

Notice there are no quotation marks within the curly braces. Within those braces, very different syntax rules apply. The `Member` property of the `StaticExtension` class is no longer an XML attribute. In terms of XML, the entire expression delimited by the curly braces is the value of the attribute, and the arguments within the curly braces appear without quotation marks.

Just like elements, markup extensions can have a `ContentProperty` attribute. Markup extensions that have only one property—such as the `StaticExtension` class with its single `Member` property—invariably mark that sole property as the content property. For markup extensions using the curly-brace syntax, this means that the `Member` property name and the equal sign can be removed:

```
<Label Text="Just some text"
       BackgroundColor="{x:Static Color.Accent}"
       TextColor="{x:Static Color.Black}"
       FontAttributes="{x:Static FontAttributes.Italic}"
       VerticalOptions="{x:Static LayoutOptions.Center}"
       HorizontalTextAlignment="{x:Static TextAlignment.Center}" />
```

This is the common form of the `x:Static` markup extension.

Obviously, the use of `x:Static` for these particular properties is unnecessary, but you can define your own static members for implementing application-wide constants, and you can reference these in your XAML files. This is demonstrated in the **SharedStatics** project.

The **SharedStatics** project contains a class named `AppConstants` that defines some constants and static fields that might be of use for formatting text:

```
namespace SharedStatics
{
    static class AppConstants
    {
        public static Color LightBackground = Color.Yellow;
        public static Color DarkForeground = Color.Blue;

        public static double NormalFontSize = 18;
        public static double TitleFontSize = 1.4 * NormalFontSize;
        public static double ParagraphSpacing = 10;

        public const FontAttributes Emphasis = FontAttributes.Italic;
        public const FontAttributes TitleAttribute = FontAttributes.Bold;

        public const TextAlignment TitleAlignment = TextAlignment.Center;
    }
}
```

You could use `Device.OnPlatform` in these definitions if you need something different for each platform.

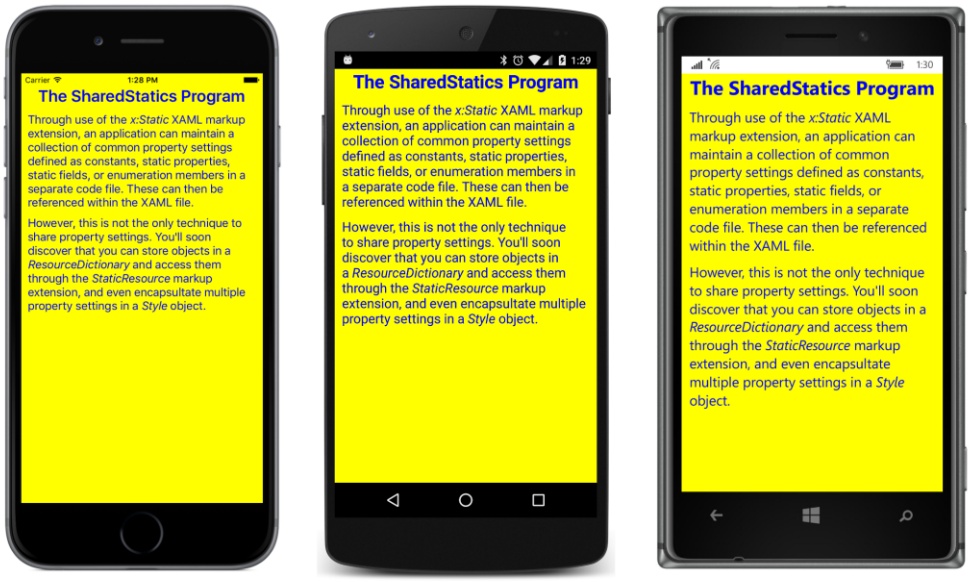
The XAML file then uses 18 `x:Static` markup extensions to reference these items. Notice the XML namespace declaration that associates the `local` prefix with the namespace of the project:

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

```
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:SharedStatics"
             x:Class="SharedStatics.SharedStaticsPage"
             BackgroundColor="{x:Static local:AppConstants.LightBackground}">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout Padding="10, 0"
                 Spacing="{x:Static local:AppConstants.ParagraphSpacing}">

        <Label Text="The SharedStatics Program"
               TextColor="{x:Static local:AppConstants.DarkForeground}"
               FontSize="{x:Static local:AppConstants.TitleFontSize}"
               FontAttributes="{x:Static local:AppConstants.TitleAttribute}"
               HorizontalTextAlignment="{x:Static local:AppConstants.TitleAlignment}" />

        <Label TextColor="{x:Static local:AppConstants.DarkForeground}"
               FontSize="{x:Static local:AppConstants.NormalFontSize}">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Through use of the " />
                    <Span Text="x:Static"
                          FontSize="{x:Static local:AppConstants.NormalFontSize}"
                          FontAttributes="{x:Static local:AppConstants.Emphasis}" />
                    <Span Text=
" XAML markup extension, an application can maintain a collection of
common property settings defined as constants, static properties or fields,
or enumeration members in a separate code file. These can then be
referenced within the XAML file." />
                </FormattedString>
            </Label.FormattedText>
        </Label>

        <Label TextColor="{x:Static local:AppConstants.DarkForeground}"
               FontSize="{x:Static local:AppConstants.NormalFontSize}">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text=
"However, this is not the only technique to share property settings.
You'll soon discover that you can store objects in a " />
                    <Span Text="ResourceDictionary"
                          FontSize="{x:Static local:AppConstants.NormalFontSize}"
                          FontAttributes="{x:Static local:AppConstants.Emphasis}" />
                    <Span Text=" and access them through the " />
                    <Span Text="StaticResource"
                          FontSize="{x:Static local:AppConstants.NormalFontSize}"
                          FontAttributes="{x:Static local:AppConstants.Emphasis}" />
                    <Span Text=
" markup extension, and even encapsultate multiple property settings in a " />
                    <Span Text="Style"
                          FontSize="{x:Static local:AppConstants.NormalFontSize}"
                          FontAttributes="{x:Static local:AppConstants.Emphasis}" />
                    <Span Text=" object." />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>
</ContentPage>
```

Each of the `Span` objects with a `FontAttributes` setting repeats the `FontSize` setting that is set on the `Label` itself because `Span` objects do not inherit font-related settings from the `Label` when another font-related setting is applied.

And here it is:



The SharedStatics Program

Through use of the *x:Static* XAML markup extension, an application can maintain a collection of common property settings defined as constants, static properties, static fields, or enumeration members in a separate code file. These can then be referenced within the XAML file.

However, this is not the only technique to share property settings. You'll soon discover that you can store objects in a *ResourceDictionary* and access them through the *StaticResource* markup extension, and even encapsultate multiple property settings in a *Style* object.

This technique allows you to use these common property settings on multiple pages, and if you ever need to change the values, you need only change the `AppSettings` file.

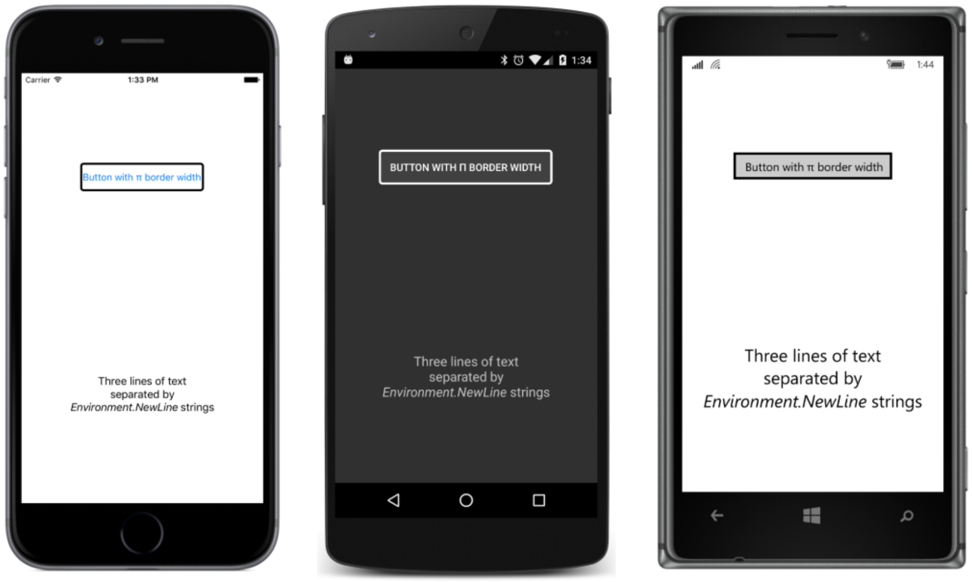
It is also possible to use `x:Static` with static properties and fields defined in classes in external libraries. The following example, named **SystemStatics,** is rather contrived—it sets the `BorderWidth` of a `Button` equal to the `PI` static field defined in the `Math` class and uses the static `Environment.NewLine` property for line breaks in text. But it demonstrates the technique.

The `Math` and `Environment` classes are both defined in the .NET `System` namespace, so a new XML namespace declaration is required to define a prefix named (for example) `sys` for `System`. Notice that this namespace declaration specifies the CLR namespace as `System` but the assembly as `mscorlib`, which originally stood for Microsoft Common Object Runtime Library but now stands for Multilanguage Standard Common Object Runtime Library:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             x:Class="SystemStatics.SystemStaticsPage">
    <StackLayout>
        <Button Text=" Button with &#x03C0; border width "
                BorderWidth="{x:Static sys:Math.PI}"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand">
            <Button.BackgroundColor>
                <OnPlatform x:TypeArguments="Color"
                            Android="#404040" />
            </Button.BackgroundColor>
            <Button.BorderColor>
                <OnPlatform x:TypeArguments="Color"
                            Android="White"
                            WinPhone="Black" />
            </Button.BorderColor>
        </Button>

        <Label VerticalOptions="CenterAndExpand"
               HorizontalTextAlignment="Center"
               FontSize="Medium">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Three lines of text" />
                    <Span Text="{x:Static sys:Environment.NewLine}" />
                    <Span Text="separated by" />
                    <Span Text="{x:Static sys:Environment.NewLine}" />
                    <Span Text="Environment.NewLine"
                          FontSize="Medium"
                          FontAttributes="Italic" />
                    <Span Text=" strings" />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>
</ContentPage>
```

The button border doesn't show up in Android unless the background color is set, and on both Android and Windows Phone the border needs a nondefault color, so some additional markup takes care of those problems. On iOS platforms, a button border tends to crowd the button text, so the text is defined with spaces at the beginning and end.

Judging solely from the visuals, we really have to take it on trust that the button border width is about 3.14 units wide, but the line breaks definitely work:


Carrier
1:33 PM
Button with π border width
Three lines of text
separated by
Environment.NewLine strings
1:34
BUTTON WITH Π BORDER WIDTH
Three lines of text
separated by
Environment.NewLine strings
1:44
Button with π border width
Three lines of text
separated by
Environment.NewLine strings

The use of curly braces for markup extensions implies that you can't display text surrounded by curly braces. The curly braces in this text will be mistaken for a markup extension:

```
<Label Text="{Text in curly braces}" />
```

That won't work. You can have curly braces elsewhere in the text string, but you can't begin with a left curly brace.

If you really need to, however, you can ensure that text is not mistaken for a XAML markup extension by beginning the text with an escape sequence that consists of a pair of left and right curly braces:

```
<Label Text="{}{Text in curly braces}" />
```

That will display the text you want.

Resource dictionaries

Xamarin.Forms also supports a second approach to sharing objects and values, and while this approach has a little more overhead than the `x:Static` markup extension, it is somewhat more versatile because everything—the shared objects and the visual elements that use them—can be expressed in XAML.

`VisualElement` defines a property named `Resources` that is of type `ResourceDictionary`—a dictionary with `string` keys and values of type `object`. Items can be added to this dictionary right in XAML, and they can be accessed in XAML with the `StaticResource` and `DynamicResource` markup extensions.

Although x:Static and StaticResource have somewhat similar names, they are quite different: x:Static references a constant, a static field, a static property, or an enumeration member, while StaticResource retrieves an object from a ResourceDictionary.

While the x:Static markup extension is intrinsic to XAML (and hence appears in XAML with an x prefix), the StaticResource and DynamicResource markup extensions are not. They were part of the original XAML implementation in the Windows Presentation Foundation, and StaticResource is also supported in Silverlight, Windows Phone 7 and 8, and Windows 8 and 10.

You'll use StaticResource for most purposes and reserve DynamicResource for some special applications, so let's begin with StaticResource.

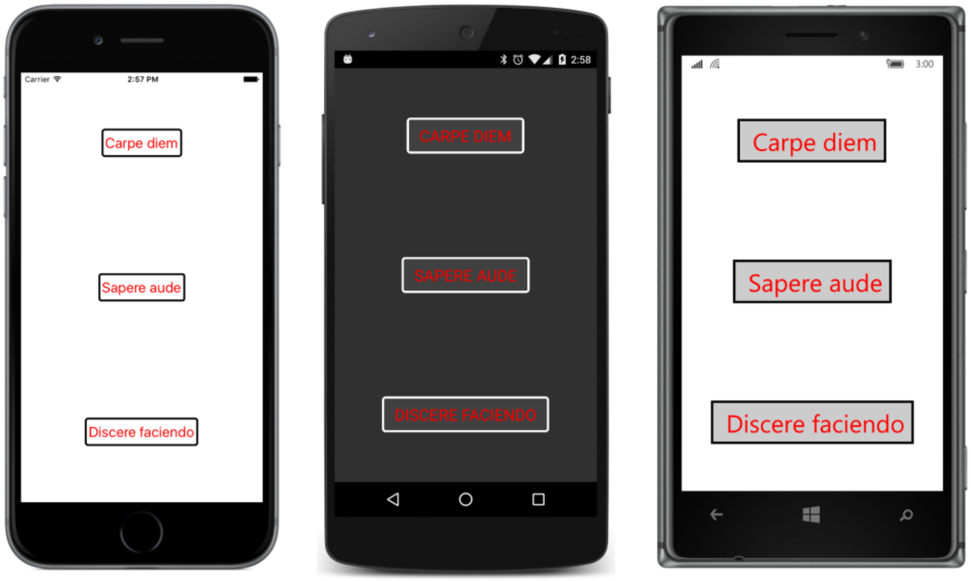
StaticResource for most purposes

Suppose you've defined three buttons in a StackLayout:

```
<StackLayout>
    <Button Text=" Carpe diem "
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
            FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>

    <Button Text=" Sapere aude "
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
                FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>

    <Button Text=" Discere faciendo "
```

```
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
            FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>
</StackLayout>
```

Of course, this is somewhat unrealistic. There are no `Clicked` events set for these buttons, and generally button text is not in Latin. But here's what they look like:



Aside from the text, all three buttons have the same properties set to the same values. Repetitious markup such as this tends to rub programmers the wrong way. It's an affront to the eye and difficult to maintain and change.

Eventually you'll see how to use styles to really cut down on the repetitious markup. For now, however, the goal is not to make the markup shorter but to consolidate the values in one place so that if you ever want to change the `TextColor` property from `Red` to `Blue`, you can do so with one edit rather than three.

Obviously, you can use x:Static for this job by defining the values in code. But let's do the whole thing in XAML by storing the values in a *resource dictionary*. Every class that derives from VisualElement has a Resources property of type ResourceDictionary. Resources that are used throughout a page are customarily stored in the Resources collection of the ContentPage.

The first step is to express the Resources property of ContentPage as a property element:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceSharing.ResourceSharingPage">

    <ContentPage.Resources>

    </ContentPage.Resources>
    …
</ContentPage>
```

If you're also defining a Padding property on the page by using property-element tags, the order doesn't matter.

For performance purposes, the Resources property is null by default, so you need to explicitly instantiate the ResourceDictionary:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceSharing.ResourceSharingPage">

    <ContentPage.Resources>
        <ResourceDictionary>

        </ResourceDictionary>
    </ContentPage.Resources>
    …
</ContentPage>
```

Between the ResourceDictionary tags, you define one or more objects or values. Each item in the dictionary must be identified with a dictionary key that you specify with the XAML x:Key attribute. For example, here's the syntax for including a LayoutOptions value in the dictionary with a descriptive key that indicates that this value is defined for setting horizontal options:

```
<LayoutOptions x:Key="horzOptions">Center</LayoutOptions>
```

Because this is a LayoutOptions value, the XAML parser accesses the LayoutOptionsConverter class to convert the content of the tags, which is the text "Center".

A second way to store a LayoutOptions value in the dictionary is to let the XAML parser instantiate the structure and set LayoutOptions properties from attributes you specify:

```
<LayoutOptions x:Key="vertOptions"
               Alignment="Center"
               Expands="True" />
```

The BorderWidth property is of type double, so the x:Double datatype element defined in the XAML 2009 specification is ideal:

```
<x:Double x:Key="borderWidth">3</x:Double>
```

You can store a Color value in the resource dictionary with a text representation of the color as content. The XAML parser uses the normal ColorTypeConverter for the text conversion:

```
<Color x:Key="textColor">Red</Color>
```

You can also specify hexadecimal ARGB values following a hash sign.

You can't initialize a Color value by setting its R, G, and B properties because those are get-only. But you can invoke a Color constructor using x:Arguments or one of the Color factory methods using x:FactoryMethod and x:Arguments.

```
<Color x:Key="textColor"
       x:FactoryMethod="FromHsla">
    <x:Arguments>
        <x:Double>0</x:Double>
        <x:Double>1</x:Double>
        <x:Double>0.5</x:Double>
        <x:Double>1</x:Double>
    </x:Arguments>
</Color>
```

Notice both the x:Key and x:FactoryMethod attributes.

The BackgroundColor and BorderColor properties of the three buttons shown above are set to values from the OnPlatform class. Fortunately you can put OnPlatform objects right in the dictionary:

```
<OnPlatform x:Key="backgroundColor"
            x:TypeArguments="Color"
            Android="#404040" />

<OnPlatform x:Key="borderColor"
            x:TypeArguments="Color"
            Android="White"
            WinPhone="Black" />
```

Notice both the x:Key and x:TypeArguments attributes.

A dictionary item for the FontSize property is somewhat problematic. The FontSize property is of type double, so if you're storing an actual numeric value in the dictionary, that's no problem. But you can't store the word "Large" in the dictionary as if it were a double. Only when a "Large" string is set to a FontSize attribute does the XAML parser use the FontSizeConverter. For that reason, you'll need to store the FontSize item as a string:

```
<x:String x:Key="fontSize">Large</x:String>
```

Here's the complete dictionary at this point:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceSharing.ResourceSharingPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions">Center</LayoutOptions>

            <LayoutOptions x:Key="vertOptions"
                           Alignment="Center"
                           Expands="True" />

            <x:Double x:Key="borderWidth">3</x:Double>

            <Color x:Key="textColor">Red</Color>

            <OnPlatform x:Key="backgroundColor"
                        x:TypeArguments="Color"
                        Android="#404040" />

            <OnPlatform x:Key="borderColor"
                        x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />

            <x:String x:Key="fontSize">Large</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>
    …
</ContentPage>
```

This is sometimes referred to as a *resources section* for the page. In real-life programming, very many XAML files begin with a resources section.

You can reference items in the dictionary by using the `StaticResource` markup extension, which is supported by `StaticResourceExtension`. The class defines a property named `Key` that you set to the dictionary key. You can use a `StaticResourceExtension` as an element within property-element tags, or you can use `StaticResourceExtension` or `StaticResource` in curly braces. If you're using the curly-brace syntax, you can leave out the `Key` and equal sign because `Key` is the content property of `StaticResourceExtension`.

The following complete XAML file in the **ResourceSharing** project illustrates three of these options:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceSharing.ResourceSharingPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions">Center</LayoutOptions>

            <LayoutOptions x:Key="vertOptions"
                           Alignment="Center"
```

```
                            Expands="True" />

            <x:Double x:Key="borderWidth">3</x:Double>

            <Color x:Key="textColor">Red</Color>

            <OnPlatform x:Key="backgroundColor"
                        x:TypeArguments="Color"
                        Android="#404040" />

            <OnPlatform x:Key="borderColor"
                        x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />

            <x:String x:Key="fontSize">Large</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Carpe diem ">
            <Button.HorizontalOptions>
                <StaticResourceExtension Key="horzOptions" />
            </Button.HorizontalOptions>

            <Button.VerticalOptions>
                <StaticResourceExtension Key="vertOptions" />
            </Button.VerticalOptions>

            <Button.BorderWidth>
                <StaticResourceExtension Key="borderWidth" />
            </Button.BorderWidth>

            <Button.TextColor>
                <StaticResourceExtension Key="textColor" />
            </Button.TextColor>

            <Button.BackgroundColor>
                <StaticResourceExtension Key="backgroundColor" />
            </Button.BackgroundColor>

            <Button.BorderColor>
                <StaticResourceExtension Key="borderColor" />
            </Button.BorderColor>

            <Button.FontSize>
                <StaticResourceExtension Key="fontSize" />
            </Button.FontSize>
        </Button>

        <Button Text=" Sapere aude "
                HorizontalOptions="{StaticResource Key=horzOptions}"
                VerticalOptions="{StaticResource Key=vertOptions}"
                BorderWidth="{StaticResource Key=borderWidth}"
                TextColor="{StaticResource Key=textColor}"
                BackgroundColor="{StaticResource Key=backgroundColor}"
                BorderColor="{StaticResource Key=borderColor}"
                FontSize="{StaticResource Key=fontSize}" />

        <Button Text=" Discere faciendo "
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                TextColor="{StaticResource textColor}"
                BackgroundColor="{StaticResource backgroundColor}"
                BorderColor="{StaticResource borderColor}"
                FontSize="{StaticResource fontSize}" />
    </StackLayout>
</ContentPage>
```

The simplest syntax in the third button is the most common, and indeed, that syntax is so ubiquitous that many longtime XAML developers might be entirely unfamiliar with the other variations. But if you use a version of `StaticResource` with the `Key` property, do not put an `x` prefix on it. The `x:Key` attribute is only for defining dictionary keys for items in the `ResourceDictionary`.

Objects and values in the dictionary are shared among all the `StaticResource` references. That's not so clear in the preceding example, but it's something to keep in mind. For example, suppose you store a `Button` object in the resource dictionary:

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Button x:Key="button"
                Text="Shared Button?"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                FontSize="Large" />
    </ResourceDictionary>
</ContentPage.Resources>
```

You can certainly use that `Button` object on your page by adding it to the `Children` collection of a `StackLayout` with the `StaticResourceExtension` element syntax:

```
<StackLayout>
    <StaticResourceExtension Key="button" />
</StackLayout>
```

However, you can't use that same dictionary item in hopes of putting another copy in the `StackLayout`:

```
<StackLayout>
    <StaticResourceExtension Key="button" />
    <StaticResourceExtension Key="button" />
</StackLayout>
```

That won't work. Both these elements reference the same `Button` object, and a particular visual element can be in only one particular location on the screen. It can't be in multiple locations.

For this reason, visual elements are not normally stored in a resource dictionary. If you need multiple elements on your page that have mostly the same properties, you'll want to use a `Style`, which is explored in Chapter 12.

A tree of dictionaries

The `ResourceDictionary` class imposes the same rules as other dictionaries: all the items in the dictionary must have keys, but duplicate keys are not allowed.

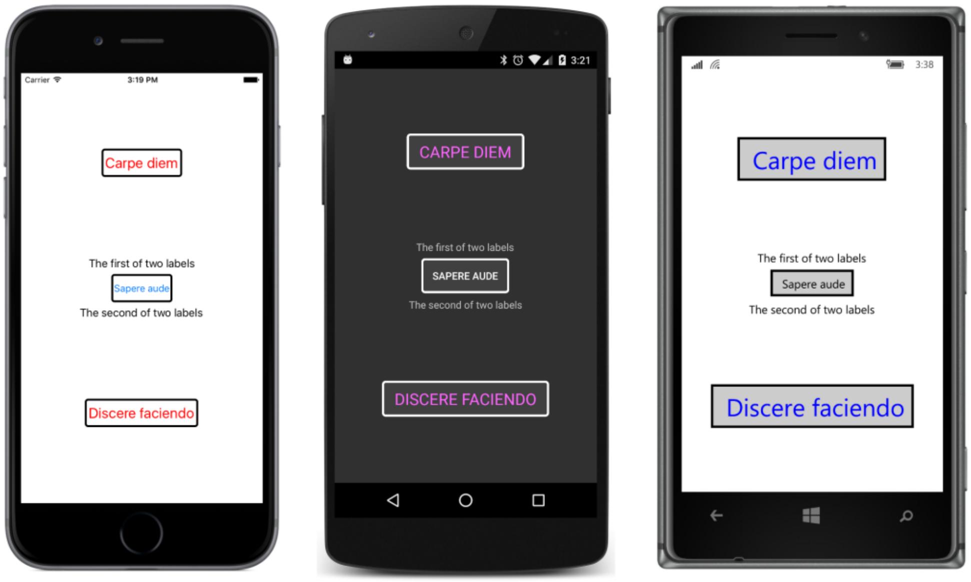
However, because every instance of `VisualElement` potentially has its own resource dictionary, your page can contain multiple dictionaries, and you can use the same keys in different dictionaries just as long as all the keys within each dictionary are unique. Conceivably, every visual element in the visual tree can have its own dictionary, but it really only makes sense for a resource dictionary to apply to multiple elements, so resource dictionaries are only commonly found defined on `Layout` or `Page` objects.

Using this technique you can construct a tree of dictionaries with dictionary keys that effectively override the keys on other dictionaries. This is demonstrated in the **ResourceTrees** project. The XAML file for the `ResourceTreesPage` class shows a `Resources` dictionary for the `ContentPage` that defines resources with keys of `horzOptions`, `vertOptions`, and `textColor`.

A second `Resources` dictionary is attached to an inner `StackLayout` for resources named `textColor` and `FontSize`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceTrees.ResourceTreesPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions">Center</LayoutOptions>

            <LayoutOptions x:Key="vertOptions"
                           Alignment="Center"
                           Expands="True" />

            <OnPlatform x:Key="textColor"
                        x:TypeArguments="Color"
                        iOS="Red"
                        Android="Pink"
                        WinPhone="Blue" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Carpe diem "
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                TextColor="{StaticResource textColor}"
                BackgroundColor="{StaticResource backgroundColor}"
                BorderColor="{StaticResource borderColor}"
                FontSize="{StaticResource fontSize}" />

        <StackLayout>
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Color x:Key="textColor">Default</Color>
                    <x:String x:Key="fontSize">Default</x:String>
                </ResourceDictionary>
            </StackLayout.Resources>

            <Label Text="The first of two labels"
                   HorizontalOptions="{StaticResource horzOptions}"
                   TextColor="{StaticResource textColor}"
                   FontSize="{StaticResource fontSize}" />

            <Button Text=" Sapere aude "
                    HorizontalOptions="{StaticResource horzOptions}"
                    BorderWidth="{StaticResource borderWidth}"
                    TextColor="{StaticResource textColor}"
                    BackgroundColor="{StaticResource backgroundColor}"
                    BorderColor="{StaticResource borderColor}"
                    FontSize="{StaticResource fontSize}" />

            <Label Text="The second of two labels"
                   HorizontalOptions="{StaticResource horzOptions}"
                   TextColor="{StaticResource textColor}"
                   FontSize="{StaticResource fontSize}" />
        </StackLayout>

        <Button Text=" Discere faciendo "
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                TextColor="{StaticResource textColor}"
                BackgroundColor="{StaticResource backgroundColor}"
                BorderColor="{StaticResource borderColor}"
                FontSize="{StaticResource fontSize}" />
    </StackLayout>
</ContentPage>
```

The `Resources` dictionary on the inner `StackLayout` applies only to items within that `StackLayout`, which are the items in the middle of this screenshot:



Here's how it works:

When the XAML parser encounters a `StaticResource` on an attribute of a visual element, it begins a search for that dictionary key. It first looks in the `ResourceDictionary` for that visual element, and if the key is not found, it looks for the key in the visual element's parent's `ResourceDictionary`, and up and up through the visual tree until it reaches the `ResourceDictionary` on the page.

But something's missing here! Where are the entries in the page's `ResourceDictionary` for `borderWidth`, `backgroundColor`, `borderColor`, and `fontSize`? They aren't in the ResourceTreesPage.xaml file!

Those items are elsewhere. The `Application` class—from which every application's `App` class derives—also defines a `Resources` property of type `ResourceDictionary`. This is handy for defining resources that apply to the entire application and not just to a particular page or layout. When the XAML parser searches up the visual tree for a matching resource key, and that key is not found in the `ResourceDictionary` for the page, it finally checks the `ResourceDictionary` defined by the `Application` class. Only if it's not found there is a `XamlParseException` raised for the `StaticResource` key-not-found error.

You can add items to your `App` class's `ResourceDictionary` object in two ways:

One approach is to add the items in code in the `App` constructor. Make sure you do this before instantiating the main `ContentPage` class:

```
public class App : Application
{
    public App()
    {
        Resources = new ResourceDictionary();
        Resources.Add("borderWidth", 3.0);
        Resources.Add("fontSize", "Large");
        Resources.Add("backgroundColor",
            Device.OnPlatform(Color.Default,
                              Color.FromRgb(0x40, 0x40, 0x40),
                              Color.Default));

        Resources.Add("borderColor",
            Device.OnPlatform(Color.Default,
                              Color.White,
                              Color.Black));

        MainPage = new ResourceTreesPage();
    }
    …
}
```

However, the `App` class can also have a XAML file of its own, and the application-wide resources can be defined in the `Resources` collection in that XAML file. To do this, you'll want to delete the App.cs file created by the Xamarin.Forms solution template. There's no template item for an `App` class, so you'll need to fake it. Add a new XAML page class—**Forms Xaml Page** in Visual Studio or **Forms ContentPage Xaml** in Xamarin Studio—to the project. Name it `App`. And immediately—before you forget—go into the App.xaml file and change the root tags to `Application`, and go into the App.xaml.cs file and change the base class to `Application`.

Now you have an `App` class that derives from `Application` and has its own XAML file. In the App.xaml file you can then instantiate a `ResourceDictionary` within `Application.Resources` property-element tags and add items to it:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ResourceTrees.App">
    <Application.Resources>
        <ResourceDictionary>
            <x:Double x:Key="borderWidth">3</x:Double>
            <x:String x:Key="fontSize">Large</x:String>

            <OnPlatform x:Key="backgroundColor"
                        x:TypeArguments="Color"
                        Android="#404040" />

            <OnPlatform x:Key="borderColor"
                        x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

The constructor in the code-behind file needs to call `InitializeComponent` to parse the App.xaml file at run time and add the items to the dictionary. This should be done prior to the normal job of instantiating the ResourceTreesPage class and setting it to the MainPage property:

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();

        MainPage = new ResourceTreesPage();
    }

    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
    }
}
```

Adding the lifecycle events is optional.

Be sure to call InitializeComponent before instantiating the page class. The constructor of the page class calls its own InitializeComponent to parse the XAML file for the page, and the StaticResource markup extensions need access to the Resources collection in the App class.

Every Resources dictionary has a particular scope: For the Resources dictionary on the App class, that scope is the entire application. A Resources dictionary on the ContentPage class applies to the whole page. A Resources dictionary on a StackLayout applies to all the children in the StackLayout. You should define and store your resources based on how you use them. Use the Resources dictionary in the App class for application-wide resources; use the Resources dictionary on the ContentPage for page-wide resources; but define additional Resources dictionaries deeper in the visual tree for resources required only in one part of the page.

As you'll see in Chapter 12, the most important items in a Resources dictionary are usually objects of type Style. In the general case, you'll have application-wide Style objects, Style objects for the page, and Style objects associated with smaller parts of the visual tree.

DynamicResource for special purposes

An alternative to StaticResource for referencing items from the Resources dictionary is DynamicResource, and if you just substitute DynamicResource for StaticResource in the example shown above, the program will seemingly run the same. However, the two markup extensions are very different. `StaticResource` accesses the item in the dictionary only once while the XAML is being parsed and the page is being built. But `DynamicResource` maintains a link between the dictionary key and the property set from that dictionary item. If the item in the resource dictionary referenced by the key changes, `DynamicResource` will detect that change and set the new value to the property.

Skeptical? Let's try it out. The **DynamicVsStatic** project has a XAML file that defines a resource item of type `string` with a key of `currentDateTime`, even though the item in the dictionary is the string "Not actually a DateTime"!

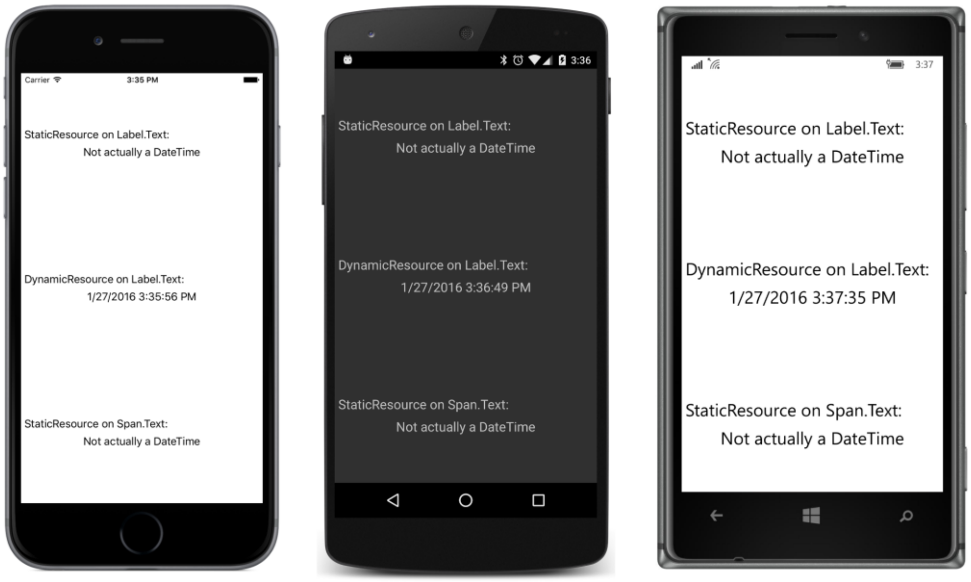
This dictionary item is referenced four times in the XAML file, but one of the references is commented out. In the first two examples, the `Text` property of a `Label` is set using `StaticResource` and `DynamicResource`. In the second two examples, the `Text` property of a `Span` object is set similarly, but the use of `DynamicResource` on the `Span` object appears in comments:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DynamicVsStatic.DynamicVsStaticPage"
             Padding="5, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <x:String x:Key="currentDateTime">Not actually a DateTime</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Label Text="StaticResource on Label.Text:"
               VerticalOptions="EndAndExpand"
               FontSize="Medium" />

        <Label Text="{StaticResource currentDateTime}"
               VerticalOptions="StartAndExpand"
               HorizontalTextAlignment="Center"
               FontSize="Medium" />

        <Label Text="DynamicResource on Label.Text:"
               VerticalOptions="EndAndExpand"
               FontSize="Medium" />

        <Label Text="{DynamicResource currentDateTime}"
               VerticalOptions="StartAndExpand"
               HorizontalTextAlignment="Center"
               FontSize="Medium" />

        <Label Text="StaticResource on Span.Text:"
               VerticalOptions="EndAndExpand"
               FontSize="Medium" />

        <Label VerticalOptions="StartAndExpand"
               HorizontalTextAlignment="Center"
               FontSize="Medium">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="{StaticResource currentDateTime}" />
                </FormattedString>
            </Label.FormattedText>
        </Label>

        <!-- This raises a run-time exception! -->

        <!--<Label Text="DynamicResource on Span.Text:"
               VerticalOptions="EndAndExpand"
               FontSize="Medium" />

        <Label VerticalOptions="StartAndExpand"
               HorizontalTextAlignment="Center"
               FontSize="Medium">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="{DynamicResource currentDateTime}" />
                </FormattedString>
            </Label.FormattedText>
        </Label>-->
    </StackLayout>
</ContentPage>
```

You'll probably expect all three of the references to the `currentDateTime` dictionary item to result in the display of the text "Not actually a DateTime". However, the code-behind file starts a timer going. Every second, the timer callback replaces that dictionary item with a new string representing an actual `DateTime` value:

```
public partial class DynamicVsStaticPage : ContentPage
{
    public DynamicVsStaticPage()
    {
        InitializeComponent();

        Device.StartTimer(TimeSpan.FromSeconds(1),
            () =>
            {
                Resources["currentDateTime"] = DateTime.Now.ToString();
                return true;
            });
    }
}
```

The result is that the `Text` properties set with `StaticResource` stay the same, while the one with `DynamicResource` changes every second to reflect the new item in the dictionary:



StaticResource on Label.Text:
Not actually a DateTime
DynamicResource on Label.Text:
1/27/2016 3:35:56 PM
StaticResource on Span.Text:
Not actually a DateTime

StaticResource on Label.Text:
Not actually a DateTime
DynamicResource on Label.Text:
1/27/2016 3:36:49 PM
StaticResource on Span.Text:
Not actually a DateTime

StaticResource on Label.Text:
Not actually a DateTime
DynamicResource on Label.Text:
1/27/2016 3:37:35 PM
StaticResource on Span.Text:
Not actually a DateTime

Here's another difference: if there is no item in the dictionary with the specified key name, `Static-Resource` will raise a run-time exception, but `DynamicResource` will not.

You can try uncommenting the block of markup at the end of the **DynamicVsStatic** project, and you will indeed encounter a run-time exception to the effect that the `Text` property could not be found. Just offhand, that exception doesn't sound quite right, but it's referring to a very real difference.

The problem is that the `Text` properties in `Label` and `Span` are defined in significantly different ways, and that difference matters a lot for `DynamicResource`. This difference will be explored in the next chapter, "The bindable infrastructure."

Lesser-used markup extensions

Three markup extensions are not used as much as the others. These are:

- `x:Null`
- `x:Type`
- `x:Array`

You use the `x:Null` extension to set a property to `null`. The syntax looks like this:

```
<SomeElement SomeProperty="{x:Null}" />
```

This doesn't make much sense unless `SomeProperty` has a default value that is not `null` when it's de-sirable to set the property to `null`. But as you'll see in Chapter 12, sometimes a property can acquire a non-`null` value from a style, and `x:Null` is pretty much the only way to override that.

The `x:Type` markup extension is used to set a property of type `Type`, the .NET class describing the type of a class or structure. Here's the syntax:

```
<AnotherElement TypeProperty="{x:Type Color}" />
```

You'll also use `x:Type` in connection with `x:Array`. The `x:Array` markup extension is always used with regular element syntax rather than curly-brace syntax. It has a required argument named `Type` that you set with the `x:Type` markup extension. This indicates the type of the elements in the array. Here's how an array might be defined in a resource dictionary:

```
<x:Array x:Key="array"
         Type="{x:Type x:String}">
    <x:String>One String</x:String>
    <x:String>Two String</x:String>
    <x:String>Red String</x:String>
    <x:String>Blue String</x:String>
</x:Array>
```

A custom markup extension

Let's create our own markup extension named `HslColorExtension`. This will allow us to set any property of type `Color` by specifying values of hue, saturation, and luminosity, but in a manner much simpler than the use of the `x:FactoryMethod` tag demonstrated in Chapter 8, "Code and XAML in harmony."

Moreover, let's put this class in a separate Portable Class Library so that you can use it from multiple applications. Such a library can be found with the other source code for this book. It's in a directory named **Libraries** that is parallel to the separate chapter directories. The name of this PCL (and the namespace of the classes within it) is **Xamarin.FormsBook.Toolkit**.

You can use this library yourself in your own applications by adding a reference to it. You can then add a new XML namespace declaration in your XAML files like so to specify this library:

```
xmlns:toolkit="clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
```

With this `toolkit` prefix you can then reference the `HslColorExtension` class in the same way you use other XAML markup extensions:

```
<BoxView Color="{toolkit:HslColor H=0.67, S=1, L=0.5}" />
```

Unlike other XAML markup extensions shown so far, this one has multiple properties, and if you're setting them as arguments with the curly-brace syntax, they must be separated with commas.

Would something like that be useful? Let's first see how to create such a library for classes that you'd like to share among applications:

In Visual Studio, from the **File** menu, select **New** and **Project**. In the **New Project** dialog, select **Visual C#** and **Cross-Platform** at the left, and **Class Library (Xamarin.Forms)** from the list. Find a location for the project and give it a name. For the PCL created for this example, the name is **Xamarin.FormsBook.Toolkit**. Click **OK**. Along with all the overhead for the project, the template creates a code file named Xamarin.FormsBook.Toolkit.cs containing a class named `Xamarin.FormsBook.Toolkit`. That's not a valid class name, so just delete that file.

In Xamarin Studio, from the **File** menu, select **New** and **Solution**. In the **New Project** dialog, select **Multiplatform** and **Library** at the left, and **Forms** and **Class Library** from the list. Find a location for it and give it a name (**Xamarin.FormsBook.Toolkit** for this example). Click **OK**. The solution template creates several files, including a file named MyPage.cs. Delete that file.

You can now add classes to this project in the normal way:

In Visual Studio, right-click the project name, select **Add** and **New Item**. In the **Add New Item** dialog, if you're just creating a code-only class, select **Visual C#** and **Code** at the left, and select **Class** from the list. Give it a name (HslColorExtension.cs for this example). Click the **Add** button.

In Xamarin Studio, in the tool menu for the project, select **Add** and **New File**. In the **New File** dialog, if you're just creating a code-only class, select **General** at the left and **Empty Class** in the list. Give it a name (HslColorExtension.cs for this example). Click the **New** button.

The **Xamarin.FormsBook.Toolkit** library will be built up and accumulate useful classes during the course of this book. But the first class in this library is `HslColorExtension`. The HslColorExtension.cs file (including the required `using` directives) looks like this:

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

namespace Xamarin.FormsBook.Toolkit
{
    public class HslColorExtension : IMarkupExtension
    {
        public HslColorExtension()
        {
            A = 1;
        }

        public double H { set; get; }

        public double S { set; get; }

        public double L { set; get; }

        public double A { set; get; }

        public object ProvideValue(IServiceProvider serviceProvider)
        {
            return Color.FromHsla(H, S, L, A);
        }
    }
}
```

Notice that the class is public, so it's visible from outside the library, and that it implements the `IMarkupExtension` interface, which means that it must include a `ProvideValue` method. However, the method doesn't make use of the `IServiceProvider` argument at all, mainly because it doesn't need to know about anything else external to itself. All it needs are the four properties to create a `Color` value, and if the `A` value isn't set, a default value of 1 (fully opaque) is used.

This **Xamarin.FormsBook.Toolkit** solution contains only a PCL project. The project can be built to generate a PCL assembly, but it cannot be run without an application that uses this assembly.

There are two ways to access this library from an application solution:

- From the PCL project of your application solution, add a reference to the library PCL assembly, which is the dynamic-link library (DLL) generated from the library project.
- Include a link to the library project from your application solution, and add a reference to that library project from the applicationt's PCL project.

The first option is necessary if you have only the DLL and not the project with source code. Perhaps you're licensing the library and don't have access to the source. But if you have access to the project, it's usually best to include a link to the library project in your solution so that you can easily make changes to the library code and rebuild the library project.

The final project in this chapter is **CustomExtensionDemo,** which makes use of the `HslColorExtension` class in the new library. The **CustomExtensionDemo** solution contains a link to the **Xamarin.FormsBook.Toolkit** PCL project, and the **References** section in the **CustomExtensionDemo** project lists the **Xamarin.FormsBook.Toolkit** assembly.

Now the application project is seemingly ready to access the library project to use the `HslColorExtension` class within the application's XAML file.

But first there's another step. Unless you've enabled XAML compilation, a reference to an external library from XAML is insufficient to ensure that the library is included with the application. The library needs to be accessed from actual code. For this reason, **Xamarin.FormsBook.Toolkit** also contains a class and method that might seem from the name to be performing important initialization for the library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class Toolkit
    {
        public static void Init()
        {
        }
    }
}
```

Whenever you use anything from this library, try to get into the habit of calling this `Init` method first thing in the `App` file:

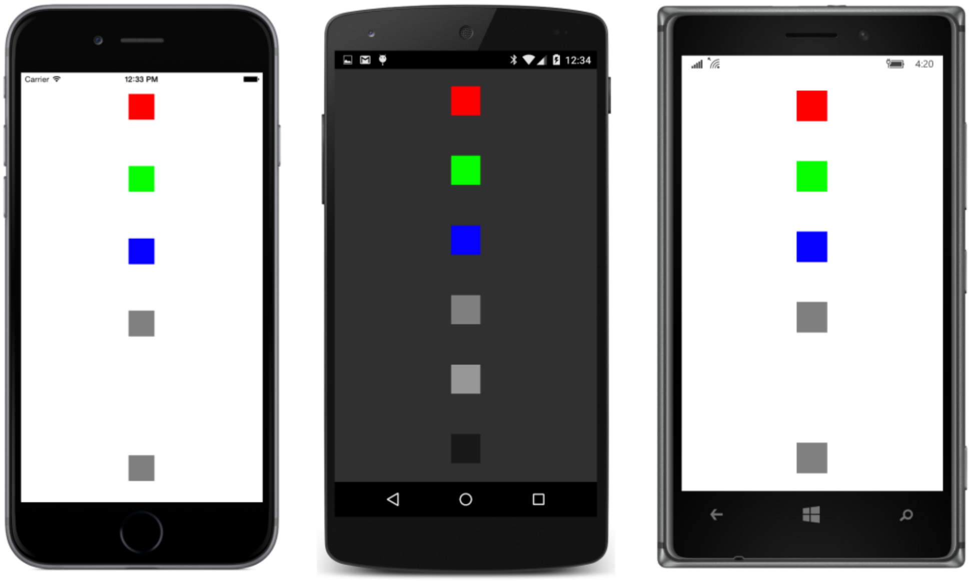
```
namespace CustomExtensionDemo
{
    public class App : Application
    {
        public App()
        {
            Xamarin.FormsBook.Toolkit.Toolkit.Init();

            MainPage = new CustomExtensionDemoPage();
        }
        …
    }
}
```

The following XAML file shows the XML namespace declaration for the **Xamarin.Forms-Book.Toolkit** library and three ways to access the custom XAML markup extension—by using an `HslColorExtension` element set with property-element syntax on the `Color` property and by using both `HslColorExtension` and `HslColor` with the more common curly-brace syntax. Again, notice the use of commas to separate the arguments within the curly braces:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="CustomExtensionDemo.CustomExtensionDemoPage">

    <StackLayout>
        <!-- Red -->
        <BoxView HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand">
            <BoxView.Color>
                <toolkit:HslColorExtension H="0" S="1" L="0.5" />
            </BoxView.Color>
        </BoxView>

        <!-- Green -->
        <BoxView HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand">
            <BoxView.Color>
                <toolkit:HslColorExtension H="0.33" S="1" L="0.5" />
            </BoxView.Color>
        </BoxView>

        <!-- Blue -->
        <BoxView Color="{toolkit:HslColor H=0.67, S=1, L=0.5}"
                 HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand" />

        <!-- Gray -->
        <BoxView Color="{toolkit:HslColor H=0, S=0, L=0.5}"
```

```
                 HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand" />

        <!-- Semitransparent white -->
        <BoxView Color="{toolkit:HslColor H=0, S=0, L=1, A=0.5}"
                 HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand" />

        <!-- Semitransparent black -->
        <BoxView Color="{toolkit:HslColor H=0, S=0, L=0, A=0.5}"
                 HorizontalOptions="Center"
                 VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The last two examples set the `A` property for 50 percent transparency, so the boxes show up as a shade of gray (or not at all) depending on the background:



Two major uses of XAML markup extensions are yet to come. In Chapter 12, you'll see the `Style` class, which is without a doubt the most popular item for including in resource dictionaries, and in Chapter 16, you'll see the powerful markup extension named `Binding`.