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:

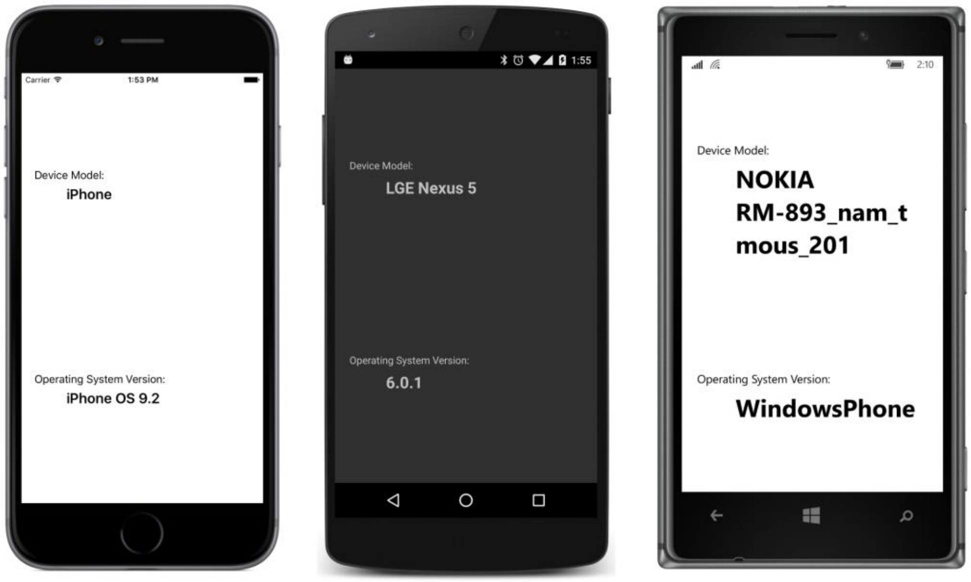
- For iOS, use the `UIDevice` class in the `UIKit` namespace.
- For Android, use various properties of the `Build` class in the `Android.OS` namespace.
- For the Windows platforms, use the `EasClientDeviceInformation` class in the `Windows.Security.ExchangeActiveSyncProvisioning` namespace.

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 defined, 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.