Chapter 22
Animation

Animation is life, action, vitality, and on computers we try to imitate those qualities despite being restricted to manipulating tiny pixels on a flat screen.

Computer animation usually refers to any type of dynamic visual change. A `Button` that simply appears on a page is not animation. But a `Button` that fades into view, or moves into place, or grows in size from a dot—that's animation. Very often, visual elements respond to user input with a change in appearance, such as a `Button` flash, a `Stepper` increment, or a `ListView` scroll. That, too, is animation.

It's sometimes desirable for an application to go beyond those automatic and conventional animations and add its own. That's what this chapter is all about.

You started seeing some of this in the previous chapter. You saw how to set transforms on visual elements and then use the timer or `Task.Delay` to animate them. Xamarin.Forms also includes its own animation infrastructure that exists in three levels of programming interfaces corresponding to the classes `ViewExtensions`, `Animation`, and `AnimationExtensions`. This animation system is versatile enough for complex jobs but exceptionally easy for simple jobs. This chapter begins with the easy high-level class (`ViewExtensions`) and then drills down to the more versatile lower levels.

The Xamarin.Forms animation classes are generally used to target properties of visual elements. A typical animation progressively changes a property from one value to another value over a period of time. The properties that are targeted by animations should be backed by bindable properties. This is not a requirement, but bindable properties are generally designed to respond to dynamic changes through the implementation of a property-changed handler. It does no good to animate a property of an object if the object doesn't even realize that the property is being changed!

There is no XAML interface for the Xamarin.Forms animation system. Consequently, all the animations in this chapter are realized in code. However, as you'll see in the next chapter, you can encapsulate animations in classes called *trigger actions* and *behaviors*, and then reference them from XAML files. Triggers and behaviors are generally the easiest way (and the recommended way) to incorporate animations within MVVM applications.

Exploring basic animations

Let's dive in with a tiny program called **AnimationTryout**. The XAML file contains nothing but a centered `Button`:

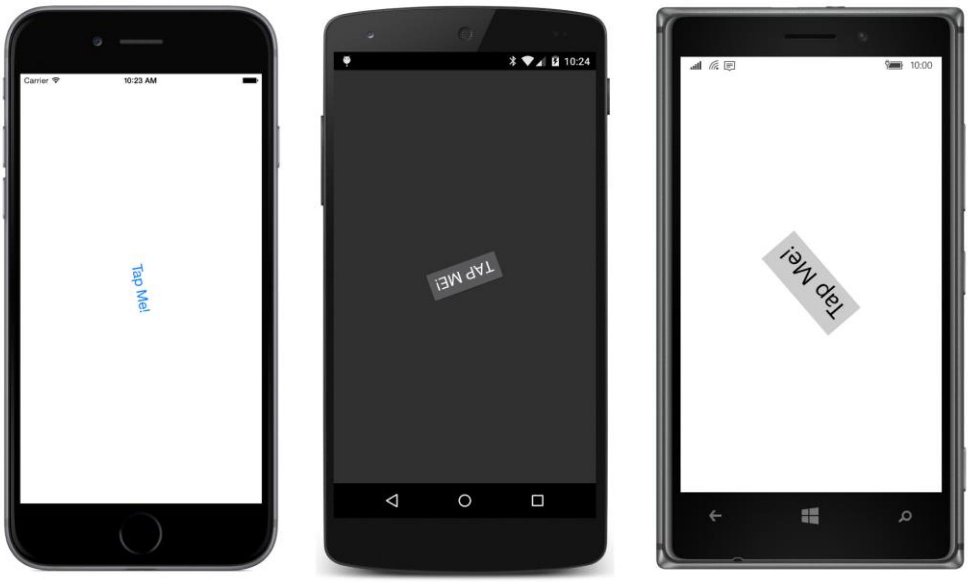
```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AnimationTryout.AnimationTryoutPage">

    <Button x:Name="button"
            Text="Tap Me!"
            FontSize="Large"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            Clicked="OnButtonClicked" />

</ContentPage>
```

For this exercise, let's ignore the actual essential function that the `Button` presumably performs within the application. In addition to wanting the button to carry out that function, suppose you'd like to spin it around in a circle when the user taps it. The `Clicked` handler in the code-behind file can do that by calling a method named `RotateTo` with an argument of 360 for the number of degrees to rotate:

```csharp
public partial class AnimationTryoutPage : ContentPage
{
    public AnimationTryoutPage()
    {
        InitializeComponent();
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        button.RotateTo(360);
    }
}
```

The `RotateTo` method is an animation that targets the `Rotation` property of `Button`. However, the `RotateTo` method is not defined in the `VisualElement` class like the `Rotation` property. It is, instead, an extension method defined in the `ViewExtensions` class.

When you run this program and tap the button, the `RotateTo` method animates the button to spin around in a full 360 degree circle. Here it is in progress:



The complete trip takes 250 milliseconds (one quarter of a second), which is the default duration of this `RotateTo` animation.

However, this program has a flaw. After you've watched the button spin around, try tapping it again. It does not rotate.

That program flaw reveals a little bit about what's going on internally: On the first call to `OnButtonClicked`, the `RotateTo` method obtains the current `Rotation` property, which is 0, and then defines an animation of the `Rotation` property from that value to the argument of `RotateTo`, which is 360. When the animation concludes after a quarter second, the `Rotation` property is left at 360.

The next time the button is pressed, the current value is 360 and the argument to `RotateTo` is also 360. Internally, the animation still occurs, but the `Rotation` property doesn't budge. It's stuck at 360.

Setting the animation duration

Here's a little variation of the `Clicked` handler in **AnimationTryout**. It doesn't fix the problem with multiple taps of the `Button`, but it does extend the animation to two seconds so you can enjoy the animation longer. The duration is specified in milliseconds as the second argument to `RotateTo`. That second argument is optional and has a default value of 250:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.RotateTo(360, 2000);
}
```

With this variation, try tapping the `Button` and then tapping it again several times as it's rotating.

You'll discover that repeated taps of the button do not send the `Rotation` property back to zero. Instead, the previous animation is cancelled and a new animation starts. But this new animation begins at whatever the `Rotation` property happens to be at the time of the tap. Each new animation still has a duration of 2 seconds, but the current `Rotation` property is closer to the end value of 360 degrees, so each new animation seems to be slower than the one before it. After the `Rotation` property finally reaches 360, however, further taps do nothing.

Relative animations

One solution to the problem of subsequent taps is to use `RelRotateTo` ("relative rotate to"), which obtains the current `Rotation` property for the start of the animation and then adds its argument to that value for the end of the animation. Here's an example:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.RelRotateTo(90, 1000);
}
```

Each tap starts an animation that rotates the button an additional 90 degrees over the course of one second. If you happen to tap the button while an animation is in progress, a new animation starts from that position, so it might end at a position that is not an increment of 90 degrees. There is no change in velocity with multiple taps because the animation is always going at the rate of 90 degrees per second.

Both `RotateTo` and `RelRotateTo` have a common underlying structure. During the course of the animation, a value is calculated—often called *t* (for time) or, sometimes, *progress*. This value is based on elapsed time and the animation's duration:

$$t = \frac{elapsedTime}{duration}$$

Values of *t* range from 0 at the beginning of the animation to 1 at the end of the animation. The animation is also defined by two values (often the values of a property), one for the start of the animation and one for the end. These are often called *start* and *end* values, or *from* and *to* values. The animation calculates a value between *from* and *to* based on a simple interpolation formula:

$$value = fromValue + t \cdot (toValue - fromValue)$$

When *t* equals 0, *value* equals *fromValue* and when *t* equals 1, *value* equals *toValue*.

Both `RotateTo` and `RelRotateTo` obtain *fromValue* from the current value of the `Rotation` property at the time the method is called. `RotateTo` sets *toValue* equal to its argument, while `RelRotateTo` sets *toValue* equal to *fromValue* plus its argument.

Awaiting animations

Another way to fix the problem with subsequent taps is to initialize the `Rotation` property prior to the call to `RotateTo`:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.Rotation = 0;
    button.RotateTo(360, 2000);
}
```

Now you can tap the `Button` again after it's stopped and it will begin the animation from the beginning. Repeated taps while the `Button` is rotating also behave differently: They start over from 0 degrees.

Interestingly, this slight variation in the code does *not* allow subsequent taps:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.RotateTo(360, 2000);
    button.Rotation = 0;
}
```

This version behaves just like the version with only the `RotateTo` method. It seems as if setting the `Rotation` property to 0 after that call does nothing.

Why doesn't it work? It doesn't work because the `RotateTo` method is asynchronous. The method returns quickly after initiating the animation, but the animation itself occurs in the background. Setting the `Rotation` property to 0 at the time the `RotateTo` method returns has no apparent effect because the setting is very quickly superseded by the background `RotateTo` animation.

Because the method is asynchronous, `RotateTo` returns a `Task` object—more specifically, a `Task<bool>` object—and that means that you can call `ContinueWith` to specify a callback function that is invoked when the animation terminates. The callback can then set the `Rotation` property back to 0 after the animation has completed:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.RotateTo(360, 2000).ContinueWith((task) =>
        {
            button.Rotation = 0;
        });
}
```

The `task` object passed to `ContinueWith` is of type `Task<bool>`, and the `ContinueWith` callback can use the `Result` property to obtain that Boolean value. The value is `true` if the animation was cancelled and `false` if it ran to completion. You can easily confirm this by displaying the return value using a `Debug.WriteLine` call and looking at the results in the **Output** window of Visual Studio or Xamarin Studio:

```
void OnButtonClicked(object sender, EventArgs args)
{
    button.RotateTo(360, 2000).ContinueWith((task) =>
        {
            System.Diagnostics.Debug.WriteLine("Cancelled? " + task.Result);
            button.Rotation = 0;
```

```
        });
}
```

If you tap the `Button` while it's being animated, you'll see `true` values returned. Every new call to `RotateTo` cancels the previous animation. If you let the animation run to completion, you'll see a `false` value returned.

It's more likely that you'll use `await` with the `RotateTo` method than `ContinueWith`:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    bool wasCancelled = await button.RotateTo(360, 2000);
    button.Rotation = 0;
}
```

Or, if you don't care about the return value:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    await button.RotateTo(360, 2000);
    button.Rotation = 0;
}
```

Notice the `async` modifier on the handler, which is required for any method that contains `await` operators.

If you've used other animation systems, it's very likely that you were required to define a callback method if you wanted the application to be notified when an animation is completed. With `await`, determining when an animation is completed—perhaps to execute some other code—becomes trivial. In this particular example the code that is executed is fairly simple, but of course it could be more complex.

Sometimes you'll want to let your animations just run to completion in the background—in which case it's not necessary to use `await` with them—and sometimes you'll want to do something when the animation has completed. But watch out: If the `Button` is also triggering some actual application function, you might not want to wait until the animation finishes before carrying that out.

`RotateTo` and `RelRotateTo` are two of several similar methods defined in the `ViewExtensions` class. Others that you'll see in this chapter include `ScaleTo`, `TranslateTo`, `FadeTo`, and `LayoutTo`. They all return `Task<bool>` objects—`false` if the animation completed without interruption and `true` if it was cancelled.

Your application can cancel one or more of these animations with a call to the static method `ViewExtensions.CancelAnimations`. Unlike all the other methods in `ViewExtensions`, this is not an extension method. You need to call it like so:

```
ViewExtensions.CancelAnimations(button);
```

That will immediately cancel all animations initiated by the extension methods in the `ViewExtensions` class that are currently running on the `button` object.

Using `await` is particularly useful for stacking sequential animations:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    await button.RotateTo(90, 250);
    await button.RotateTo(-90, 500);
    await button.RotateTo(0, 250);
}
```

The total animation defined here requires one second. The `Button` swings 90 degrees clockwise in the first quarter second, then 180 degrees counterclockwise in the next half second, and then 90 degrees clockwise to end up at 0 degrees again. You need `await` on the first two so that they're sequential, but you don't need it on the third if there's nothing else to execute in the `Clicked` handler after the third animation has completed.

A composite animation like this is often known as a *key-frame animation*. You are specifying a series of rotation angles and times, and the overall animation is interpolating between those. In most animation systems, key-frame animations are often more difficult to use than simple animations. But with `await`, key-frame animations become trivial.

The return value of `Task<bool>` does not necessarily indicate that the animation is running in a secondary thread. In fact, at least part of the animation—the part that actually sets the `Rotation` property—must run in the user-interface thread. It is theoretically possible for the entire animation to run in the user-interface thread. As you saw in the previous chapter, animations that you create with `Device.StartTimer` or `Task.Delay` run entirely in the user-interface thread, although the underlying timer mechanism might involve a secondary thread.

You'll see later in this chapter how an animation method can still return a `Task` object but run entirely in the user-interface thread. This technique allows code to use timers for pacing animations but still provide a structured `Task`-based notification when the code has completed.

Composite animations

You can mix awaited and nonawaited calls to create composite animations. For example, suppose you want the button to spin around 360 degrees at the same time it expands in size and then contracts.

The `ViewExtensions` class defines a method name `ScaleTo` that animates the `Scale` property just as `RotateTo` animates the `Rotate` property. The expansion and contraction of the `Button` size requires two sequential animations, but these should occur at the same time as the rotation, which only requires one call. For that reason, the `RotateTo` call can execute without an `await`, and while that animation is running in the background, the method can make two sequential calls to `ScaleTo`. Try this in **AnimationTryout**:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    button.Rotation = 0;
    button.RotateTo(360, 2000);
    await button.ScaleTo(5, 1000);
```

```
    await button.ScaleTo(1, 1000);
}
```

The durations are made somewhat longer than they would be normally so that you can see what's happening. The RotateTo method returns immediately, and the first ScaleTo animation begins at that time. But that await operator on the first ScaleTo delays the call of the second ScaleTo until the first ScaleTo has completed. At that time, the RotateTo animation is only half finished and the Button has rotated 180 degrees. During the next 1,000 milliseconds, that RotateTo completes at about the same time the second ScaleTo animation completes.

Here's the Button as it's making its way through the animation:

Because the OnButtonClicked method is flagged with the async keyword and the first RotateTo does not have an await operator, you'll get a warning message from the compiler that states: "Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call."

If you prefer not to see that warning message, you can turn it off with a #pragma statement that disables that particular warning:

```
#pragma warning disable 4014
```

You could place that statement at the top of your source code file to disable warnings throughout the file. Or you can place it before the offending call and reenable those warnings after the call by using:

```
#pragma warning restore 4014
```

Task.WhenAll and Task.WhenAny

Another powerful option is available that lets you combine animations in a very structured way without worrying about compiler warnings. The static `Task.WhenAll` and `Task.WhenAny` methods of the `Task` class are intended to run multiple asynchronous methods concurrently. Each of these methods can accept an array or other collection of multiple arguments, each of which is a method that returns a `Task` object. The `Task.WhenAll` and `Task.WhenAny` methods also return `Task` objects. The `WhenAll` method completes when all the methods in its collection have completed. The `WhenAny` method completes when any method in its collection completes execution while the other methods in the `WhenAny` collection continue to run.

Watch out: The `Task` class also includes static methods named `WaitAll` and `WaitAny`. You don't want to use those methods. They block the user-interface thread until the task or tasks have completed.

Because the `Task.WhenAll` and `Task.WhenAny` methods themselves return `Task` objects, you can use `await` with them. Here's one way to implement the composite animation shown above without any compiler warnings: The `Task.WhenAny` call contains two tasks, the first of which runs for two seconds and the second runs for one second. When that second task completes, the `Task.WhenAny` call also completes. The `RotateTo` method is still running, but now the second `ScaleTo` method can start:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    button.Rotation = 0;

    await Task.WhenAny<bool>
        (
            button.RotateTo(360, 2000),
            button.ScaleTo(5, 1000)
        );
    await button.ScaleTo(1, 1000);
}
```

You can also use `Task.Delay` with these methods to introduce little delays into the composite animation.

Rotation and anchors

The `AnchorX` and `AnchorY` properties set the center of scaling or rotation for the `Scale` and `Rotation` properties, so they also affect the `ScaleTo` and `RotateTo` animations.

The **CircleButton** program rotates a `Button` in a circle, but not like the code you've seen previously. This program rotates a `Button` around the center of the screen, and for that it requires `AnchorX` and `AnchorY`.

The XAML file puts the `Button` in an `AbsoluteLayout`:

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

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

    <AbsoluteLayout x:Name="absoluteLayout"
                    SizeChanged="OnSizeChanged">
        <Button x:Name="button"
                Text="Tap Me!"
                FontSize="Large"
                SizeChanged="OnSizeChanged"
                Clicked="OnButtonClicked" />
    </AbsoluteLayout>
</ContentPage>
```

The only reason this program uses an `AbsoluteLayout` for the `Button` is to place the `Button` precisely at a particular location on the screen. The XAML file sets the same `SizeChanged` handler on both the `AbsoluteLayout` and the `Button`. That event handler saves the center of the `AbsoluteLayout` as the `Point` field named `center` and also saves the distance from that center to the nearest edge as the `radius` field:

```
public partial class CircleButtonPage : ContentPage
{
    Point center;
    double radius;

    public CircleButtonPage()
    {
        InitializeComponent();
    }

    void OnSizeChanged(object sender, EventArgs args)
    {
        center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
        radius = Math.Min(absoluteLayout.Width, absoluteLayout.Height) / 2;
        AbsoluteLayout.SetLayoutBounds(button,
            new Rectangle(center.X - button.Width / 2, center.Y - radius,
                          AbsoluteLayout.AutoSize,
                          AbsoluteLayout.AutoSize));
    }
    …
}
```

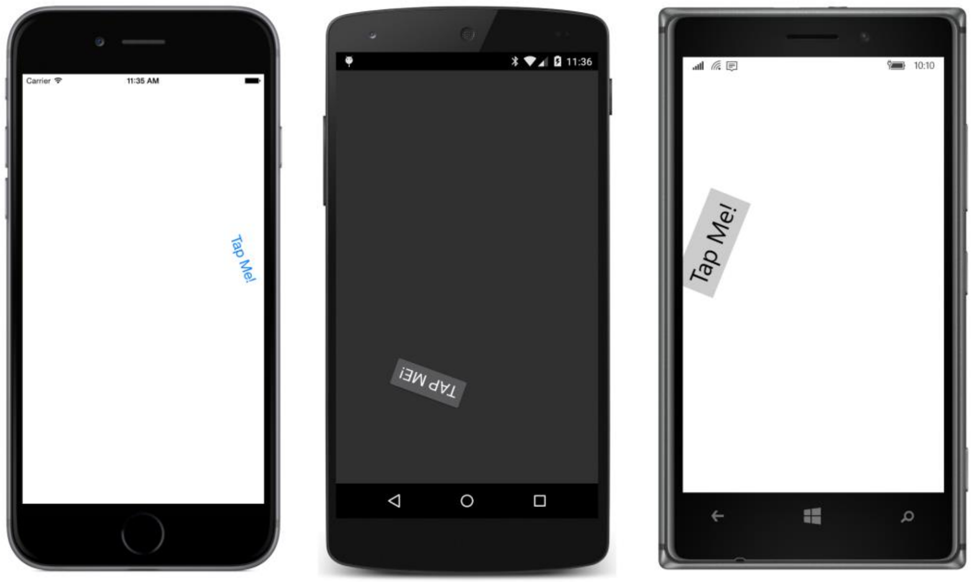
The `OnSizeChanged` handler concludes by positioning the `Button` in the horizontal center of the page, but with its top edge a distance of `radius` above the center of the `AbsoluteLayout`:

Recall that the AnchorX and AnchorY properties must be set to numbers that are relative to the width and height of the Button. An AnchorX value of 0 refers to the left edge of the Button and a value of 1 refers to the right edge. Similarly, an AnchorY value of 0 refers to the top of the Button and a value of 1 refers to the bottom.

To rotate this Button around the point saved as center, AnchorX and AnchorY must be set to values based on the center point. The center of the Button is directly above the center of the page, so the default 0.5 value of AnchorX is fine. AnchorY, however, needs a value from the top of the Button to the center point, but in units of the button's height:

```
public partial class CircleButtonPage : ContentPage
{
    …
    async void OnButtonClicked(object sender, EventArgs args)
    {
        button.Rotation = 0;
        button.AnchorY = radius / button.Height;
        await button.RotateTo(360, 1000);
    }
}
```

The Button then makes a full rotation of 360 degrees around the center of the page. Here it is in progress:



Easing functions

You've already seen the following key-frame animation that swings the `Button` one way and then the other:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    await button.RotateTo(90, 250);
    await button.RotateTo(-90, 500);
    await button.RotateTo(0, 250);
}
```

But the animation doesn't quite look right. The movement seems very mechanical and robotic because the rotations have a constant angular velocity. Shouldn't the `Button` at least slow down as it reverses direction and then speed up again?

You can control velocity changes in animations with the use of easing functions. You already saw a couple of homemade easing functions in Chapter 21, "Transforms." Xamarin.Forms includes an `Easing` class that allows you to specify a simple transfer function that controls how animations speed up or slow down as they're running.

You'll recall that animations generally involve a variable named *t* or *progress* that increases from 0 to 1 over the course of the animation. This *t* variable is then used in an interpolation between *from* and *to* values:

$$value = fromValue + t \cdot (toValue - fromValue)$$

The easing function introduces a little transfer function into this calculation:

$$value = fromValue + EasingFunc(t) \cdot (toValue - fromValue)$$

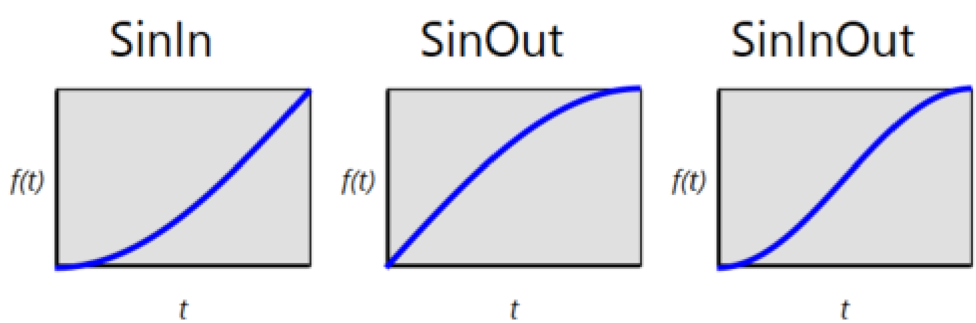
The `Easing` class defines a method named `Ease` that performs this job. For an input of 0, the `Ease` method returns 0, and for an input of 1, `Ease` returns 1. Between those two values, some mathematics—often a rather *tiny* chunk of mathematics—gives the animation a nonconstant velocity. (As you'll see later, it's not entirely necessary that the `Ease` method maps 0 to 0 and 1 to 1, but that's certainly the normal case.)

You can define your own easing functions, but the `Easing` class defines 11 static read-only fields of type `Easing` for your convenience:

- `Linear` (the default)
- `SinIn`, `SinOut`, and `SinInOut`
- `CubicIn`, `CubicOut`, and `CubicInOut`
- `BounceIn` and `BounceOut`
- `SpringIn` and `SpringOut`

The `In` and `Out` suffixes indicate whether the effect is prominent at the beginning of the animation, at the end, or both.

The `SinIn`, `SinOut`, and `SinInOut` easing functions are based on sine and cosine functions:



In each of these graphs, the horizontal axis is linear time, left to right from 0 to 1. The vertical axis shows the output of the `Ease` method, 0 to 1 from bottom to top. A steeper, more vertical slope is faster, while a more horizontal slope is slower.

The `SinIn` is the first quadrant of a cosine curve but subtracted from 1 so it goes from 0 to 1; it starts off slow but gets faster. The `SinOut` is the first quadrant of a sine curve, starting off somewhat faster than a linear animation but slowing down toward the end. The `SinInOut` is the first half of a cosine curve (again adjusted to go from 0 to 1); it's slow at the beginning and the end.

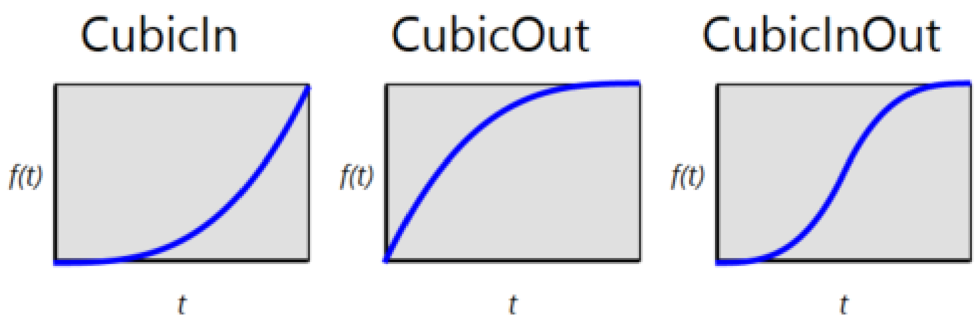
Because harmonic motion is best described by sine curves, these easing functions are ideal for a

`Button` swinging to and fro. You can specify an object of type `Easing` as the last argument to the `RotateTo` methods:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    await button.RotateTo(90, 250, Easing.SinOut);
    await button.RotateTo(-90, 500, Easing.SinInOut);
    await button.RotateTo(0, 250, Easing.SinIn);
}
```

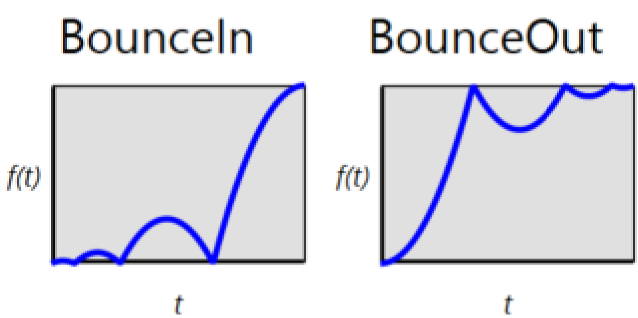
And now the movement is much more natural. The `Button` slows down as it approaches the point when it reverses movement and then speeds up again.

The `CubicIn` easing function is simply the input raised to the third power. The `CubicOut` is the reverse of that, and `CubicInOut` combines the two effects:



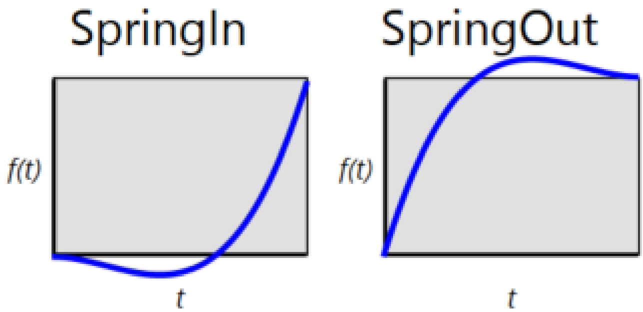
The difference in velocity is more accentuated than the sine easing.

The `BounceIn` and `BounceOut` bounce at the beginning or end, respectively:



As you might imagine, the `BounceOut` is great for animating transforms that seem to come up against an obstacle.

The output of the `SpringIn` and `SpringOut` functions actually go beyond the range of 0 to 1. The `SpringIn` has an output that drops below 0 initially, and the `SpringOut` output goes beyond the value of 1:



In other animation systems, these `SpringIn` and `SpringOut` patterns are usually known as *back-ease* functions, and you saw the underlying mathematics in the **BoxViewClock** sample in the previous chapter. In fact, you can rewrite the `Convert` method in `SecondBackEaseConverter` like this and it will work the same:

```
public object Convert(object value, Type targetType,
                      object parameter, CultureInfo culture)
{
    int seconds = (int)((double)value / 6);     // 0, 1, 2, ... 60
    double t = (double)value / 6 % 1;           // 0 --> 1
    double v = 0;                               // 0 --> 1

    if (t < 0.5)
    {
        v = 0.5 * Easing.SpringIn.Ease(2 * t);
    }
    else
    {
        v = 0.5 * (1 + Easing.SpringOut.Ease(2 * (t - 0.5)));
    }

    return 6 * (seconds + v);
}
```

There is no `SpringInOut` object, so the `Convert` method must break each second into two halves. When `t` is less than 0.5, the `SpringIn` object is applied. However, the input to the `Ease` method needs to be doubled to range from 0 to 1, and the output needs to be halved to range from 0 to 0.5. The `SpringOut` call must be adjusted likewise: When `t` ranges from 0.5 to 1, the input to the `Ease` method needs to range from 0 to 1, and the output needs to be adjusted to range from 0.5 to 1.

Let's try some more easing functions. The **BounceButton** program has a XAML file that is the same as **AnimationTryout,** and the `Clicked` handler for the `Button` has just three statements:

```
public partial class BounceButtonPage : ContentPage
{
    public BounceButtonPage()
```

```
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        await button.TranslateTo(0, (Height - button.Height) / 2, 1000, Easing.BounceOut);
        await Task.Delay(2000);
        await button.TranslateTo(0, 0, 1000, Easing.SpringOut);
    }
}
```

The `TranslateTo` method animates the `TranslationX` and `TranslationY` properties. The first two arguments are named `x` and `y`, and they indicate the final values to be set to `TranslationX` and `TranslationY`. The first `TranslateTo` call here does not move the `Button` horizontally, so the first argument is 0. The second argument is the distance between the bottom of the `Button` and the bottom of the page. The `Button` is vertically centered on the page, so that distance is half the height of the page minus half the height of the `Button`.

That first animation is performed in 1,000 milliseconds. Then there's a two-second delay, and the `Button` is translated back to its original position with `x` and `y` arguments of 0. The second `TranslateTo` animation uses the `Easing.SpringOut` function, so you probably expect the `Button` to overshoot its mark and then settle back into its final position.

However, the `TranslateTo` method clamps the output of any easing function that goes outside the range of 0 to 1. Later on in this chapter you'll see a fix for that flaw in the `TranslateTo` method.

Your own easing functions

It's easy to make your own easing functions. All that's required is a method of type `Func<double, double>`, which is a function with a `double` argument and a `double` return value. This is a transfer function: It should return 0 for an argument of 0, and 1 for an argument of 1. But between those two values, anything goes.

Generally you'll define a custom easing function as the argument to the `Easing` constructor. That's the only constructor `Easing` defines, but the `Easing` class also defines an implicit conversion from a `Func<double, double>` to `Easing`.

The Xamarin.Forms animation functions call the `Ease` method of the `Easing` object. That `Ease` method also has a `double` argument and a `double` return value, and it basically provides public access to the easing function you specify in the `Easing` constructor. (The graphs earlier in this chapter that showed the various predefined easing functions were generated by a program that accessed the `Ease` methods of the various predefined `Easing` objects.)

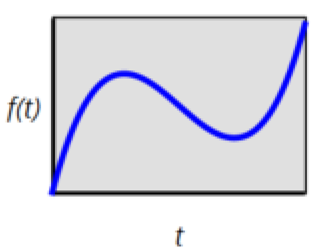
Here's a program that incorporates two custom easing functions to control the scaling of a `Button`. These functions somewhat contradict the meaning of the word "ease," which is why the program is called **UneasyScale**. The first of these two easing functions truncates the incoming value to the discrete values 0, 0.2, 0.4, 0.6, 0.8, and 1, so the `Button` increases in size in jumps. The `Button` is then decreased in size with another easing function that applies a little random variation to the incoming value.

The first of these easing functions is specified as a lambda function argument to the `Easing` constructor. The second is a method cast to an `Easing` object:

```
public partial class UneasyScalePage : ContentPage
{
    Random random = new Random();

    public UneasyScalePage()
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        double scale = Math.Min(Width / button.Width, Height / button.Height);
        await button.ScaleTo(scale, 1000, new Easing(t => (int)(5 * t) / 5.0));
        await button.ScaleTo(1, 1000, (Easing)RandomEase);
    }

    double RandomEase(double t)
    {
        return t == 0 || t == 1 ? t : t + 0.25 * (random.NextDouble() - 0.5);
    }
}
```

Unfortunately, it's easier to make disjointed functions like these rather than smoother and more interesting transfer functions. Those tend to be necessarily a bit more complex.

For example, suppose you want an easing function that looks like this:



It starts off fast, then slows down and reverses course, but then reverses course again to rise quickly into the final stretch.

You might guess that this is a polynomial equation, or at least that it can be approximated by a polynomial equation. It has two points where the slope is zero, which further suggests that this is a cubic and can be represented like this:

$$f(t) = a \cdot t^3 + b \cdot t^2 + c \cdot t + d$$

Now all we need to find are values of *a*, *b*, *c*, and *d* that will cause the transfer function to behave as we want.

For the endpoints, we know that:

$$f(0) = 0$$
$$f(1) = 1$$

This means that:

$$d = 0$$

and:

$$1 = a + b + c$$

If we say further that the two dips in the curve are at *t* equal to 1/3 and 2/3, and the values of *f(t)* at those points are 2/3 and 1/3, respectively, then:

$$\frac{2}{3} = a \cdot \frac{1}{27} + b \cdot \frac{1}{9} + c \cdot \frac{1}{3}$$

$$\frac{1}{3} = a \cdot \frac{8}{27} + b \cdot \frac{4}{9} + c \cdot \frac{2}{3}$$

Those two equations are somewhat more readable and manipulable if they are converted to integer coefficients, so what we have are three equations with three unknowns:

$$1 = a + b + c$$

$$18 = a + 3 \cdot b + 9 \cdot c$$

$$9 = 8 \cdot a + 12 \cdot b + 18 \cdot c$$

And with a little manipulation and combination and work, you can find *a*, *b*, and *c*:

$$a = 9$$

$$b = -\frac{27}{2}$$

$$c = \frac{11}{2}$$

Let's see if it does what we think it will do. The **CustomCubicEase** program has a XAML file that is the same as the previous projects. The easing function is here expressed directly as a `Func<double, double>` object so that it can be conveniently used in two `ScaleTo` calls. The `Button` is first scaled up in size, and then after a one-second pause, the `Button` is scaled back to normal:

```
public partial class CustomCubicEasePage : ContentPage
```

```
{
    public CustomCubicEasePage()
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        Func<double, double> customEase = t => 9 * t * t * t - 13.5 * t * t + 5.5 * t;

        double scale = Math.Min(Width / button.Width, Height / button.Height);
        await button.ScaleTo(scale, 1000, customEase);
        await Task.Delay(1000);
        await button.ScaleTo(1, 1000, customEase);
    }
}
```

If you don't consider the job of making your own easing functions to be "fun and relaxing," one good source for many standard easing functions is the website http://robertpenner.com/easing/.

It's also possible to construct easing functions from `Math.Sin` and `Math.Cos` if you need simple harmonic motion and to combine those with `Math.Exp` for exponential increases or decay.

Let's take an example: Suppose you want a `Button` that, when clicked, swings down from its lower-left corner, almost as if the `Button` were a picture attached to a wall with a couple of nails, and one of the nails falls out, so the picture slips down and hangs by a single nail in its lower-left corner.

You can follow along with this exercise in the **AnimationTryout** program. In the `Clicked` handler for the `Button`, let's begin by setting the `AnchorX` and `AnchorY` properties and then call `RotateTo` for a 90-degree swing:

```
button.AnchorX = 0;
button.AnchorY = 1;
await button.RotateTo(90, 3000);
```

Here's the result when that animation has completed:

But this really cries out for an easing function so that the `Button` swings back and forth a bit from that corner before settling. To begin, let's first add a do-nothing linear easing function to the `RotateTo` call:

```
await button.RotateTo(90, 3000, new Easing(t => t));
```

Let's now add some sinusoidal behavior. That's either a sine or a cosine. We want the swing to be slow at the beginning, so that would imply a cosine rather than a sine. Let's set the argument to the `Math.Cos` method so that as `t` goes from 0 to 1, the angle is 0 through 10π. That's five complete cycles of the cosine curve, which means that the `Button` swings five times back and forth:

```
await button.RotateTo(90, 3000, new Easing(t => Math.Cos(10 * Math.PI * t)));
```

Of course, this is not right at all. When `t` is zero, the `Math.Cos` method returns 1, so the animation starts off by jumping to a value of 90 degrees. For subsequent values of `t`, the `Math.Cos` function returns values ranging from 1 through –1, so the `Button` swings five times from 90 degrees to –90 degrees and back to 90 degrees, finally coming to a rest at 90 degrees. That is indeed where we want the animation to end, but we want the animation to start at 0 degrees

Nevertheless, let's ignore that problem for a moment. Let's instead tackle what initially seems to be the more complex problem. We don't want the `Button` to swing a full 180 degrees five times. We want the swings of the `Button` to decay over time before it comes to rest.

There's an easy way to do that. We can multiply the `Math.Cos` method by a `Math.Exp` call with a negative argument based on `t`:

```
Math.Exp(-5 * t)
```

The `Math.Exp` method raises the mathematical constant *e* (approximately 2.7) to the specified power.

When `t` is 0 at the beginning of the animation, *e* to the 0 power is 1. And when `t` is 1, *e* to the negative fifth power is less than .01, which is very close to zero. (You don't need to use –5 in this call; you can experiment to find a value that seems best.)

Let's multiply the `Math.Cos` result by the `Math.Exp` result:

```
await button.RotateTo(90, 3000, new Easing(t => Math.Cos(10 * Math.PI * t) * Math.Exp(-5 * t)));
```

We are very very close. The `Math.Exp` does indeed damp the `Math.Cos` call, but the product is backward The product is 1 when `t` is 0 and nearly 0 when `t` is 1. Can we fix this by simply subtracting the whole expression from 1? Let's try it:

```
await button.RotateTo(90, 3000,
    new Easing(t => 1 - Math.Cos(10 * Math.PI * t) * Math.Exp(-5 * t)));
```

Now the easing function properly returns 0 when `t` is 0, and close enough to 1 when `t` is 1.

And, what's more important, the easing function is now visually satisfactory as well. It really looks as if the `Button` drops from its mooring and swings several times before coming to rest.

Let's now call `TranslateTo` to make the `Button` drop off and fall to the bottom of the page. How far does the `Button` need to drop?

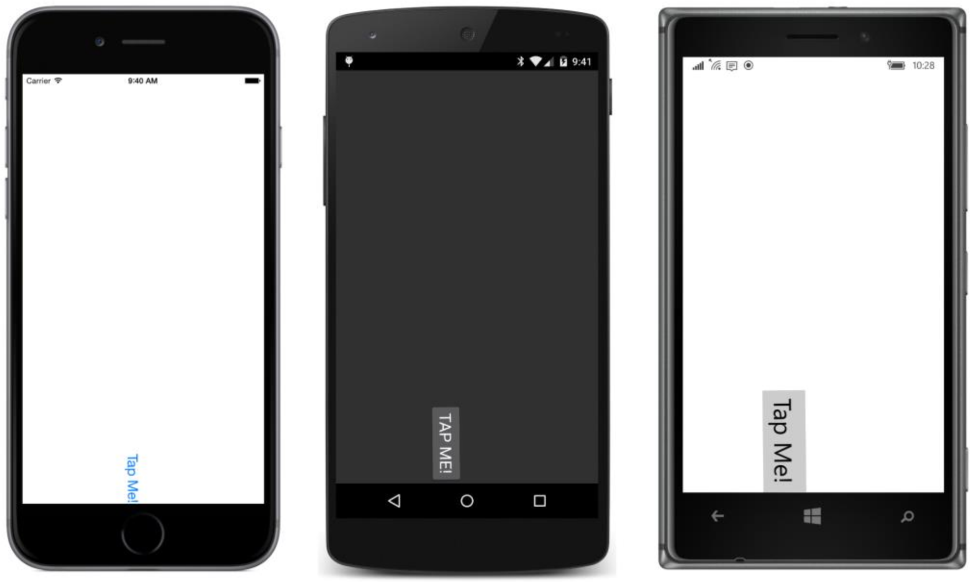
The `Button` was originally positioned in the center of the page. That means that the distance between the bottom of the `Button` and the page was half the height of the page minus the height of the `Button`:

```
(Height - button.Height) / 2
```

But now the `Button` has swung 90 degrees from its lower-left corner, so the `Button` is closer to the bottom of the page by its width. Here's the full call to `TranslateTo` to drop the `Button` to the bottom of the page and make it bounce a little:

```
await button.TranslateTo(0, (Height - button.Height) / 2 - button.Width,
                         1000, Easing.BounceOut);
```

The `Button` comes to rest like this:

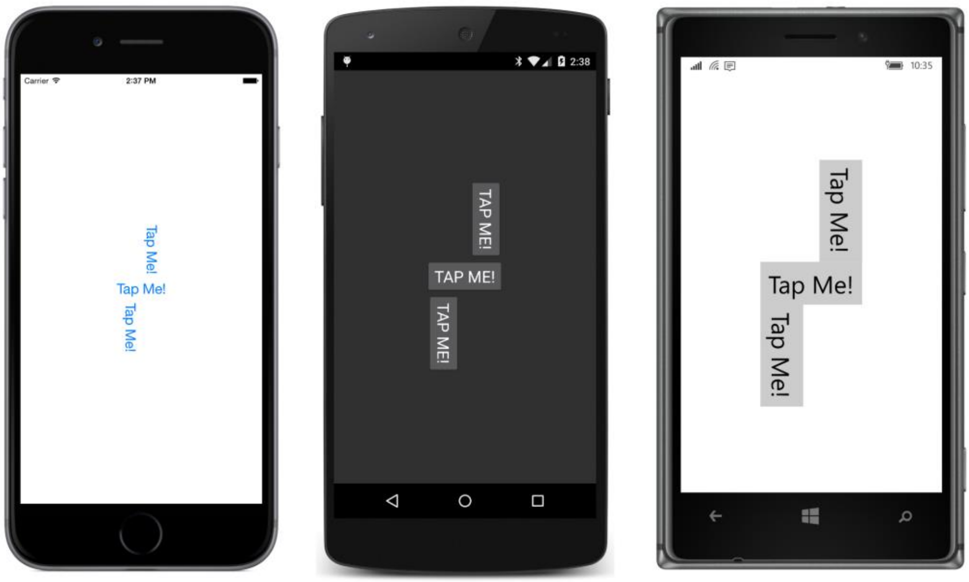


Now let's make the `Button` keel over and land upside down, which means that we want to rotate the `Button` around the upper-right corner. This requires a change in the `AnchorX` and `AnchorY` properties:

```
button.AnchorX = 1;
button.AnchorY = 0;
```

But that's a problem—a *big* problem—because a change in the `AnchorX` and `AnchorY` properties actually changes the location of the `Button`. Try it! The `Button` suddenly leaps up and to the right. Where the `Button` jumps to is exactly the position it would be if the first `RotateTo` had been based on these new `AnchorX` and `AnchorY` values—a rotation around its upper-right corner rather than its lower-left corner.

Can you visualize that? Here's a little mockup that shows the original position of the `Button`, the `Button` rotated 90 degrees clockwise from its lower-left corner, and the `Button` rotated 90 degrees clockwise from its upper-right corner:



When we set new values of `AnchorX` and `AnchorY`, we need to adjust the `TranslationX` and `TranslationY` properties so that the `Button` essentially moves from the rotated position in the upper-right to the rotated position in the lower-left. `TranslationX` needs to be decreased by the width of the `Button` and then increased by its height. `TranslationY` needs to be increased by both the height of the `Button` and the width of the `Button`. Let's try that:

```
button.TranslationX -= button.Width - button.Height;
button.TranslationY += button.Width + button.Height;
```

And that preserves the position of the `Button` when the `AnchorX` and `AnchorY` properties are changed to the button's upper-right corner.

Now the `Button` can be rotated around its upper-right corner as it falls over, with another little bounce, of course:

```
await button.RotateTo(180, 1000, Easing.BounceOut);
```

And now the `Button` can ascend up the screen and simultaneously fade out:

```
await Task.WhenAll
    (
        button.FadeTo(0, 4000),
        button.TranslateTo(0, -Height, 5000, Easing.CubicIn)
    );
```

The `FadeTo` method animates the `Opacity` property, in this case from its default value of 1 to the value 0 specified as the first argument.

Here's the complete program, called **SwingButton** (referring to the first animation) and concluding with a restoration of the `Button` to its original position so that you can try it again:

```
public partial class SwingButtonPage : ContentPage
{
    public SwingButtonPage()
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        // Swing down from lower-left corner.
        button.AnchorX = 0;
        button.AnchorY = 1;

        await button.RotateTo(90, 3000,
            new Easing(t => 1 - Math.Cos(10 * Math.PI * t) * Math.Exp(-5 * t)));

        // Drop to the bottom of the screen.
        await button.TranslateTo(0, (Height - button.Height) / 2 - button.Width,
                                 1000, Easing.BounceOut);

        // Prepare AnchorX and AnchorY for next rotation.
        button.AnchorX = 1;
        button.AnchorY = 0;

        // Compensate for the change in AnchorX and AnchorY.
        button.TranslationX -= button.Width - button.Height;
        button.TranslationY += button.Width + button.Height;

        // Fall over.
        await button.RotateTo(180, 1000, Easing.BounceOut);

        // Fade out while ascending to the top of the screen.
        await Task.WhenAll
            (
                button.FadeTo(0, 4000),
                button.TranslateTo(0, -Height, 5000, Easing.CubicIn)
            );

        // After three seconds, return the Button to normal.
        await Task.Delay(3000);
        button.TranslationX = 0;
        button.TranslationY = 0;
        button.Rotation = 0;
        button.Opacity = 1;
    }
}
```

An easing function is supposed to return 0 when the input is 0 and 1 when the input is 1, but it's possible to break these rules, and sometimes that makes sense. For example, suppose you want an animation that moves an element a little—perhaps it vibrates it in some way—but the animation should return the element to its original position at the end. For something like this it makes sense for the easing function to return 0 when the input is both 0 and 1, but something other than 0 between those values.

This is the idea behind `JiggleButton`, which is in the **Xamarin.FormsBook.Toolkit** library. `JiggleButton` derives from `Button` and installs a `Clicked` handler for the sole purpose of jiggling the button when you click it:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class JiggleButton : Button
    {
        bool isJiggling;

        public JiggleButton()
        {
            Clicked += async (sender, args) =>
                {
                    if (isJiggling)
                        return;

                    isJiggling = true;

                    await this.RotateTo(15, 1000, new Easing(t =>
                                                Math.Sin(Math.PI * t) *
                                                Math.Sin(Math.PI * 20 * t)));
                    isJiggling = false;
                };
        }
    }
}
```

The `RotateTo` method seems to rotate the button by 15 degrees over the course of one second. However, the custom `Easing` object has a different idea. It consists solely of the product of two sine functions. As `t` goes from 0 to 1, the first `Math.Sin` function sweeps the first half of a sine curve, so it goes from 0 when `t` is 0, to 1 when `t` is 0.5, and back to 0 when `t` is 1.

The second `Math.Sin` call is the jiggle part. As `t` goes from 0 to 1, this call goes through 10 cycles of a sine curve. Without the first `Math.Sin` call, this would rotate the button from 0 to 15 degrees, then to –15 degrees, and back to 0 ten times. But the first `Math.Sin` call dampens that rotation at the beginning and end of the animation, allowing only a full 15 and –15 degree rotation in the middle.

A little code involving the `isJiggling` field protects the `Clicked` handler from starting a new animation when one is already in progress. This is an advantage of using `await` with the animation methods: You know exactly when the animation is completed.

The **JiggleButtonDemo** XAML file creates three `JiggleButton` objects so that you can play with them:

```
<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="JiggleButtonDemo.JiggleButtonDemoPage">
```

```
    <StackLayout>
        <toolkit:JiggleButton Text="Tap Me!"
                              FontSize="Large"
                              HorizontalOptions="Center"
                              VerticalOptions="CenterAndExpand" />

        <toolkit:JiggleButton Text="Tap Me!"
                              FontSize="Large"
                              HorizontalOptions="Center"
                              VerticalOptions="CenterAndExpand" />

        <toolkit:JiggleButton Text="Tap Me!"
                              FontSize="Large"
                              HorizontalOptions="Center"
                              VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

Entrance animations

One common type of animation in real-life programming occurs when a page is first made visible. The various elements on the page can be animated briefly before settling into their final states. This is often called an *entrance animation* and can involve:

- Translation, to move elements into their final positions.
- Scale, to enlarge or shrink elements to their final sizes.
- Changes in `Opacity` to fade elements into view.
- 3D rotation to make it seem as if a whole page swings into view.

Generally you'll want the elements on the page to come to rest with default values of these properties: `TranslationX` and `TranslationY` values of 0, `Scale` and `Opacity` values of 1, and all `Rotation` properties set to 0.

In other words, the entrance animations should *end* at each property's default value, which means that they begin at nondefault values. This approach also allows the program to apply other transforms to these elements at a later time without taking the entrance animations into account.

When designing the layout in XAML you'll want to simply ignore these animations. As an example, here is a page with several elements solely for demonstration purposes. The program is called **FadingEntrance**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FadingEntrance.FadingEntrancePage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="10, 20, 10, 10"
                    Android="10"
                    WinPhone="10" />
```

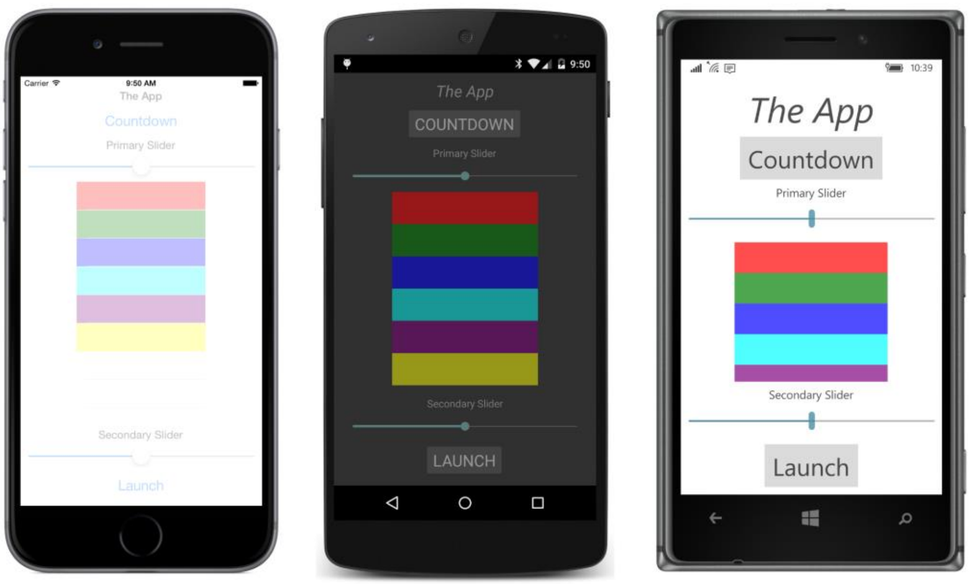
```
    </ContentPage.Padding>

    <StackLayout x:Name="stackLayout">
        <Label Text="The App"
               Style="{DynamicResource TitleStyle}"
               FontAttributes="Italic"
               HorizontalOptions="Center" />

        <Button Text="Countdown"
                FontSize="Large"
                HorizontalOptions="Center" />

        <Label Text="Primary Slider"
               HorizontalOptions="Center" />

        <Slider Value="0.5" />

        <ListView HorizontalOptions="Center"
                  WidthRequest="200">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type Color}">
                    <Color>Red</Color>
                    <Color>Green</Color>
                    <Color>Blue</Color>
                    <Color>Aqua</Color>
                    <Color>Purple</Color>
                    <Color>Yellow</Color>
                </x:Array>
            </ListView.ItemsSource>

            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <BoxView Color="{Binding}" />
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

        <Label Text="Secondary Slider"
               HorizontalOptions="Center" />

        <Slider Value="0.5" />

        <Button Text="Launch"
                FontSize="Large"
                HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

The code-behind file overrides the `OnAppearing` method. The `OnAppearing` method is called after the page is laid out but before the page becomes visible. All the elements on the page have been sized and positioned, so if you need to obtain that information you can do so during this method. In the

FadingEntrance program, the `OnAppearing` override sets the `Opacity` property of the `StackLayout` to 0 (thus making everything within the `StackLayout` invisible) and then animates it to 1:

```
public partial class FadingEntrancePage : ContentPage
{
    public FadingEntrancePage()
    {
        InitializeComponent();
    }

    protected override void OnAppearing()
    {
        base.OnAppearing();

        stackLayout.Opacity = 0;
        stackLayout.FadeTo(1, 3000);
    }
}
```

Here's the page in the process of fading into view:



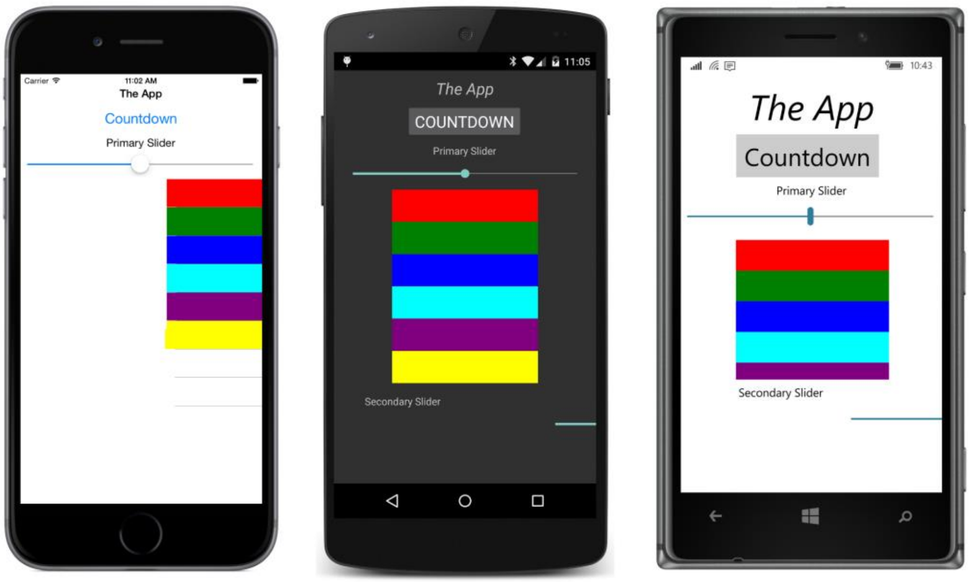
Let's try another. The XAML file in the **SlidingEntrance** program is the same as **FadingEntrance**, but the `OnAppearing` override begins by setting all the `TranslationX` properties of the children of the `StackLayout` to alternating values of 1000 and –1000:

```
public partial class SlidingEntrancePage : ContentPage
{
    public SlidingEntrancePage()
    {
        InitializeComponent();
    }
```

```
    async protected override void OnAppearing()
    {
        base.OnAppearing();

        double offset = 1000;

        foreach (View view in stackLayout.Children)
        {
            view.TranslationX = offset;
            offset *= -1;
        }

        foreach (View view in stackLayout.Children)
        {
            await Task.WhenAny(view.TranslateTo(0, 0, 1000, Easing.SpringOut),
                               Task.Delay(100));
        }
    }
}
```

The second `foreach` loop then animates these children back to the default settings of `TranslationX` and `TranslationY`. However, the animations are staggered and overlapped. Here's how: The first call to `Task.WhenAny` starts the first `TranslateTo` animation, which completes after one second. However, the second argument to `Task.WhenAny` is `Task.Delay`, which completes in one-tenth of a second, and that's when `Task.WhenAny` also completes. The `foreach` loop fetches the next child, which then begins its own one-second animation. Every animation begins one-tenth of a second after the previous one.

Here's the result in process:



The `TranslateTo` call uses the `Easing.SpringOut` function, which means that each animated element should overshoot its destination and then move backward to come at rest in the center of the page. However, you won't see this happen. As you've already discovered, the `TranslateTo` method stops working when an easing function has an output that exceeds 1.

You'll see a solution for this—and a version of this program with elements that do overshoot their destinations—later in this chapter.

Finally, here's a **SwingingEntrance** animation:

```
public partial class SwingingEntrancePage : ContentPage
{
    public SwingingEntrancePage()
    {
        InitializeComponent();
    }

    async protected override void OnAppearing()
    {
        base.OnAppearing();

        stackLayout.AnchorX = 0;
        stackLayout.RotationY = 180;
        await stackLayout.RotateYTo(0, 1000, Easing.CubicOut);
        stackLayout.AnchorX = 0.5;
    }
}
```

The `RotateYTo` method rotates the entire `StackLayout` and its children around the Y axis from 180 degrees to 0 degrees. With an `AnchorX` setting of 0, the rotation is actually around the left edge of the `StackLayout`. The `StackLayout` won't be visible until the `RotationY` value is less than 90 degrees, but the result looks a little better if the rotation starts before the page actually becomes visible. The `CubicOut` easing function causes the animation to slow down as it nears completion. Here it is in progress:

After the animation has completed, the `OnAppearing` method returns `AnchorX` to its original value so that everything has default values for any future animations that the program might want to implement.

Forever animations

At the opposite extreme from entrance animations are *forever animations*. An application can implement an animation that goes on "forever," or at least until the program ends. Often the sole purpose of such animations is to demonstrate the capabilities of an animation system, but preferably in a delightful or amusing manner.

The first example is called **FadingTextAnimation** and uses `FadeTo` to fade two `Label` elements in and out. The XAML file puts both `Label` elements in a single-cell `Grid` so that they overlap. The second one has its `Opacity` property set to 0:

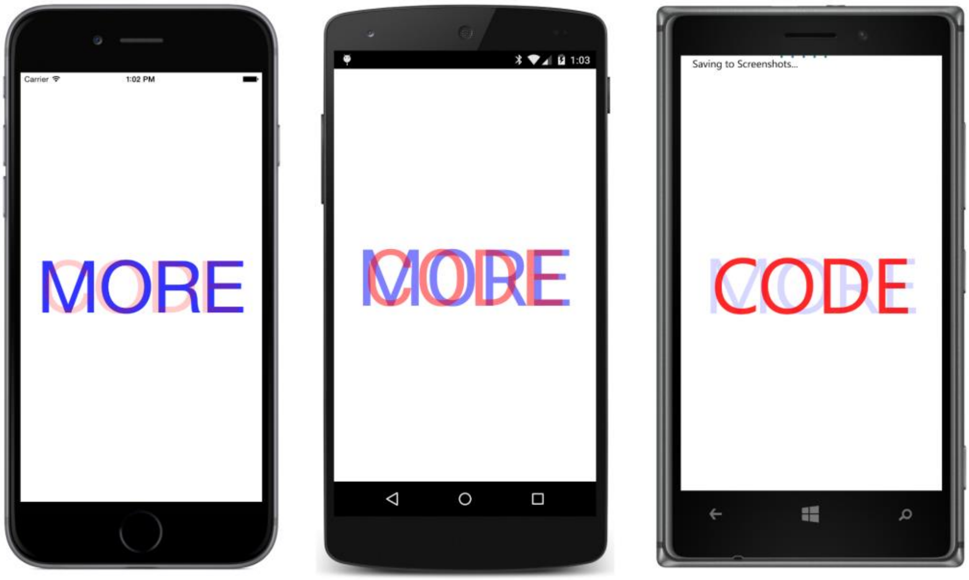
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FadingTextAnimation.FadingTextAnimationPage"
             BackgroundColor="White"
             SizeChanged="OnPageSizeChanged">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
```

```
                <Setter Property="HorizontalTextAlignment" Value="Center" />
                <Setter Property="VerticalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid>
        <Label x:Name="label1"
               Text="MORE"
               TextColor="Blue" />

        <Label x:Name="label2"
               Text="CODE"
               TextColor="Red"
               Opacity="0" />
    </Grid>
</ContentPage>
```

One simple way to create an animation that runs “forever” is to put all your animation code—using `await` of course—within a `while` loop with a condition of `true`. Then call that method from the constructor:

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

        // Start the animation going.
        AnimationLoop();
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        if (Width > 0)
        {
            double fontSize = 0.3 * Width;
            label1.FontSize = fontSize;
            label2.FontSize = fontSize;
        }
    }

    async void AnimationLoop()
    {
        while (true)
        {
            await Task.WhenAll(label1.FadeTo(0, 1000),
                               label2.FadeTo(1, 1000));

            await Task.WhenAll(label1.FadeTo(1, 1000),
                               label2.FadeTo(0, 1000));
        }
    }
}
```

Infinite loops are usually dangerous, but this one executes very briefly once every second when the `Task.WhenAll` method signals a completion of the two animations—the first fading out one `Label` and the second fading in the other `Label`. The `SizeChanged` handler for the page sets the `FontSize` of the text, so the text approaches the width of the page:



Does it mean "More code" or "Code more"? Perhaps both.

Here's another animation that targets text. The **PalindromeAnimation** program spins individual characters 180 degress to turn them upside down. Fortunately, the characters comprise a palindrome that reads the same forward and backward:

When all the characters are flipped upside down, the whole collection of characters is flipped, and the animation starts again.

The XAML file simply contains a horizontal `StackLayout`, without any children just yet:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PalindromeAnimation.PalindromeAnimationPage"
             SizeChanged="OnPageSizeChanged">

    <StackLayout x:Name="stackLayout"
                 Orientation="Horizontal"
                 HorizontalOptions="Center"
                 VerticalOptions="Center"
                 Spacing="0" />
</ContentPage>
```

The constructor of the code-behind file fills this `StackLayout` with 17 `Label` elements to spell out the palindromic phrase “NEVER ODD OR EVEN.” As in the previous program, the `SizeChanged` handler for the page adjusts the size of these labels. Each `Label` is given a uniform `WidthRequest` and a `FontSize` based on that width. Each character in the text string must occupy the same width so that they are still spaced the same when they flip upside down:

```
public partial class PalindromeAnimationPage : ContentPage
{
    string text = "NEVER ODD OR EVEN";
    double[] anchorX = { 0.5, 0.5, 0.5, 0.5, 1, 0,
                         0.5, 1, 1, -1,
                         0.5, 1, 0,
                         0.5, 0.5, 0.5, 0.5 };
```

```
public PalindromeAnimationPage()
{
    InitializeComponent();

    // Add a Label to the StackLayout for each character.
    for (int i = 0; i < text.Length; i++)
    {
        Label label = new Label
        {
            Text = text[i].ToString(),
            HorizontalTextAlignment = TextAlignment.Center
        };
        stackLayout.Children.Add(label);
    }

    // Start the animation.
    AnimationLoop();
}

void OnPageSizeChanged(object sender, EventArgs args)
{
    // Adjust the size and font based on the display width.
    double width = 0.8 * this.Width / stackLayout.Children.Count;

    foreach (Label label in stackLayout.Children.OfType<Label>())
    {
        label.FontSize = 1.4 * width;
        label.WidthRequest = width;
    }
}

async void AnimationLoop()
{
    bool backwards = false;

    while (true)
    {
        // Let's just sit here a second.
        await Task.Delay(1000);

        // Prepare for overlapping rotations.
        Label previousLabel = null;

        // Loop through all the labels.
        IEnumerable<Label> labels = stackLayout.Children.OfType<Label>();

        foreach (Label label in backwards ? labels.Reverse() : labels)
        {
            uint flipTime = 250;

            // Set the AnchorX and AnchorY properties.
            int index = stackLayout.Children.IndexOf(label);
            label.AnchorX = anchorX[index];
            label.AnchorY = 1;
```

```
                if (previousLabel == null)
                {
                    // For the first Label in the sequence, rotate it 90 degrees.
                    await label.RelRotateTo(90, flipTime / 2);
                }
                else
                {
                    // For the second and subsequent, also finish the previous flip.
                    await Task.WhenAll(label.RelRotateTo(90, flipTime / 2),
                                       previousLabel.RelRotateTo(90, flipTime / 2));
                }

                // If it's the last one, finish the flip.
                if (label == (backwards ? labels.First() : labels.Last()))
                {
                    await label.RelRotateTo(90, flipTime / 2);
                }

                previousLabel = label;
            }

            // Rotate the entire stack.
            stackLayout.AnchorY = 1;
            await stackLayout.RelRotateTo(180, 1000);

            // Flip the backwards flag.
            backwards ^= true;
        }
    }
}
```

Much of the complexity of the `AnimationLoop` method results from overlapping animations. Each letter needs to rotate by 180 degrees. However, the final 90 degrees of each letter rotation overlaps with the first 90 degrees of the next letter. This requires that the first letter and the last letter be handled differently.

The letter rotations are further complicated by the settings of the `AnchorX` and `AnchorY` properties. For each rotation, `AnchorY` is set to 1 and the rotation occurs around the bottom of the `Label`. But the setting of the `AnchorX` property depends on where the letter occurs in the phrase. The first four letters of "NEVER" can spin around the bottom center of the letter because they form the word "EVEN" when inverted. But the "R" needs to spin around its lower-right corner so that it becomes the end of the word "OR". The space after "NEVER" needs to spin around its lower-left corner so that it becomes the space between "OR" and "EVEN". Essentially, the "R" of "NEVER" and the space swap places. The rest of the phrase continues similarly. The various `AnchorX` values for each letter are stored in the `anchorX` array at the top of the class.

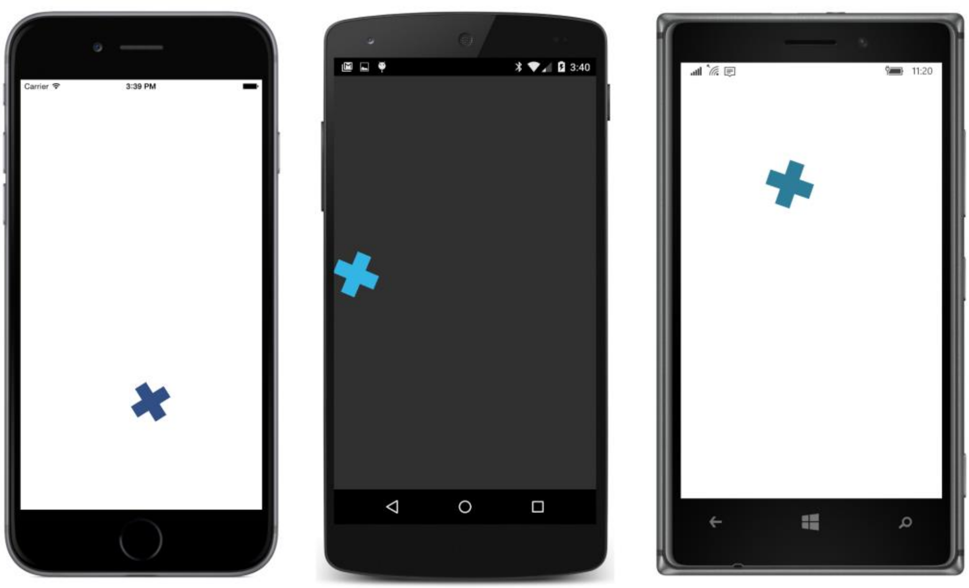
When all the letters have been individually rotated, then the whole `StackLayout` is rotated by 180 degrees. Although that rotated `StackLayout` looks the same as the `StackLayout` when the program started running, it is not the same. The last letter of the phrase is now the first child in the StackLayout and the first letter is now the last child in the StackLayout. That's the reason for the backwards variable. The foreach statement uses that to enumerate through the StackLayout children in a forward or backward direction.

You'll notice that all the AnchorX and AnchorY properties are set in the AnimationLoop right before the animation is started, even though they never change over the course of the program. This is to accommodate the problem with iOS. The properties must be set after the element has been sized, and setting those properties within this loop is simply convenient.

If that problem with iOS did not exist, all the AnchorX and AnchorY properties could be set in the program's constructor or even in the XAML file. It's not unreasonable to define all 17 Label elements in the XAML file with unique AnchorX settings on each Label and the common AnchorY setting in a Style.

As it is, on iOS devices, the **PalindromeAnimation** program cannot survive a change in orientation from portrait to landscape and back. After the Label elements are resized, there is nothing the application can do to fix the internal use of the AnchorX and AnchorY properties.

The **CopterAnimation** program simulates a little helicopter flying in a circle around the page. The simulation, however, is very simple: The helicopter is simply two BoxView elements sized and arranged to look like wings:



The program has two continuous rotations. The fast one spins the helicopter's blades around its center. A slower rotation moves the wing assemblage in a circle around the center of the page. Both rotations use the default AnchorX and AnchorY settings of 0.5, so there's no problem on iOS.

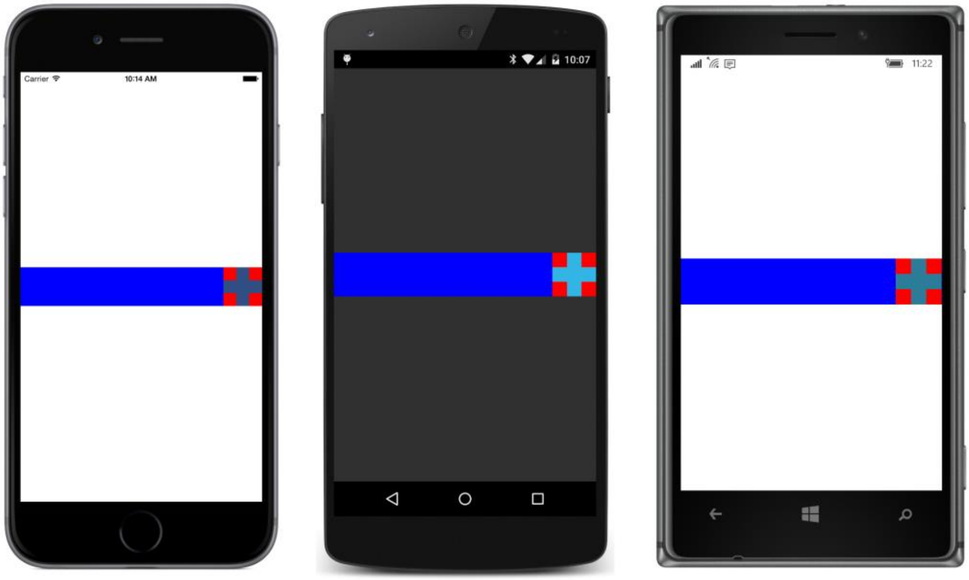
However, the program implicitly uses the width of the phone for the circumference of the circle that the copter wings fly around. If you turn the phone sideways to landscape mode, the copter will actually fly outside the bounds of the phone.

The secret to the simplicity of **CopterAnimation** is the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="CopterAnimation.CopterAnimationPage">
    <ContentView x:Name="revolveTarget"
                 HorizontalOptions="Fill"
                 VerticalOptions="Center">
        <ContentView x:Name="copterView"
                     HorizontalOptions="End">
            <AbsoluteLayout>
                <BoxView AbsoluteLayout.LayoutBounds="20, 0, 20, 60"
                         Color="Accent" />

                <BoxView AbsoluteLayout.LayoutBounds="0, 20, 60, 20"
                         Color="Accent" />
            </AbsoluteLayout>
        </ContentView>
    </ContentView>
</ContentPage>
```

The entire layout consists of two nested `ContentView` elements, with an `AbsoluteLayout` in the inner `ContentView` for the two `BoxView` wings. The outer `ContentView` (named `revolveTarget`) extends to the width of the phone and is vertically centered on the page, but it is only as tall as the inner `ContentView`. The inner `ContentView` (named `copterView`) is positioned at the far right of the outer `ContentView`.

You can probably visualize this more easily if you turn off the animation and give the two `ContentView` elements different background colors, for example, blue and red:

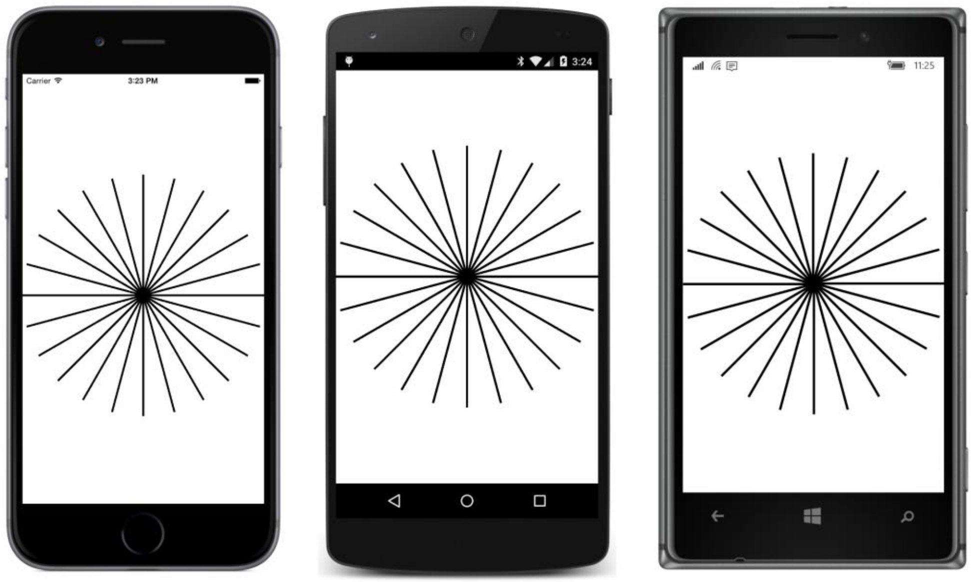


Now you can see fairly easily that both these `ContentView` elements can be rotated around their centers to achieve the effect of rotating wings flying in a circle:

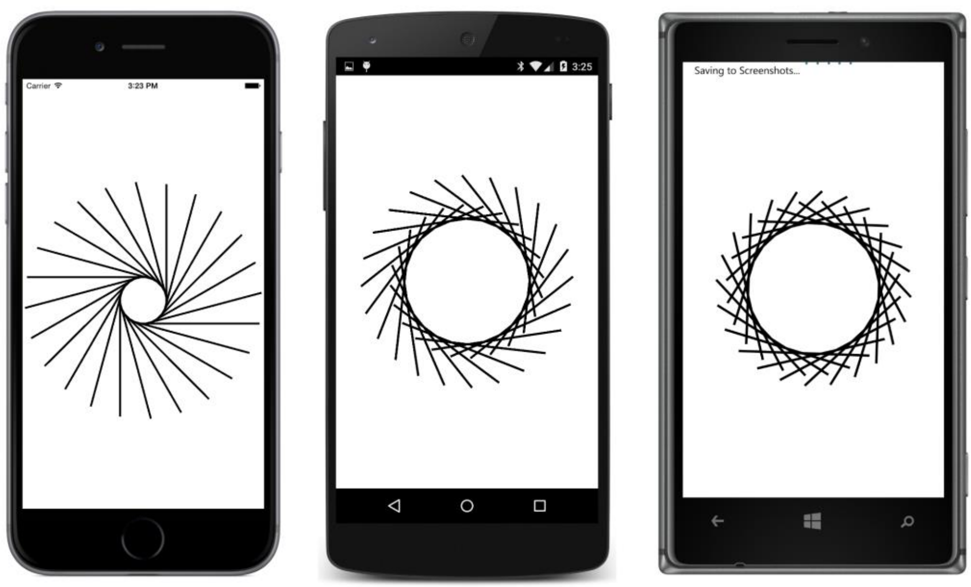
```
public partial class CopterAnimationPage : ContentPage
{
    public CopterAnimationPage()
    {
        InitializeComponent();

        AnimationLoop();
    }

    async void AnimationLoop()
    {
        while (true)
        {
            revolveTarget.Rotation = 0;
            copterView.Rotation = 0;

            await Task.WhenAll(revolveTarget.RotateTo(360, 5000),
                               copterView.RotateTo(360 * 5, 5000));
        }
    }
}
```

Both animations have a duration of five seconds, but during that time, the outer `ContentView` rotates only once around its center while the copter wing assembly rotates five times around its center.

The **RotatingSpokes** program draws 24 spokes emanating from the center of the page with a length based on the lesser of the height and width of the page. Of course, each of the spokes is a thin `BoxView` element:



After three seconds, the assemblage of spokes begins to rotate around the center. That goes on for a little while, and then each individual spoke begins rotating around *its* center, making an interesting changing pattern:



As with **CopterAnimation**, the **RotatingSpokes** program uses default values of `AnchorX` and `AnchorY` for all the rotations, so there's no problem changing the phone orientation on iOS devices.

But the XAML file in **RotatingSpokes** consists solely of an `AbsoluteLayout` and suggests nothing about how the program works:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="RotatingSpokes.RotatingSpokesPage"
             BackgroundColor="White"
             SizeChanged="OnPageSizeChanged">
    <AbsoluteLayout x:Name="absoluteLayout"
                    HorizontalOptions="Center"
                    VerticalOptions="Center" />
</ContentPage>
```

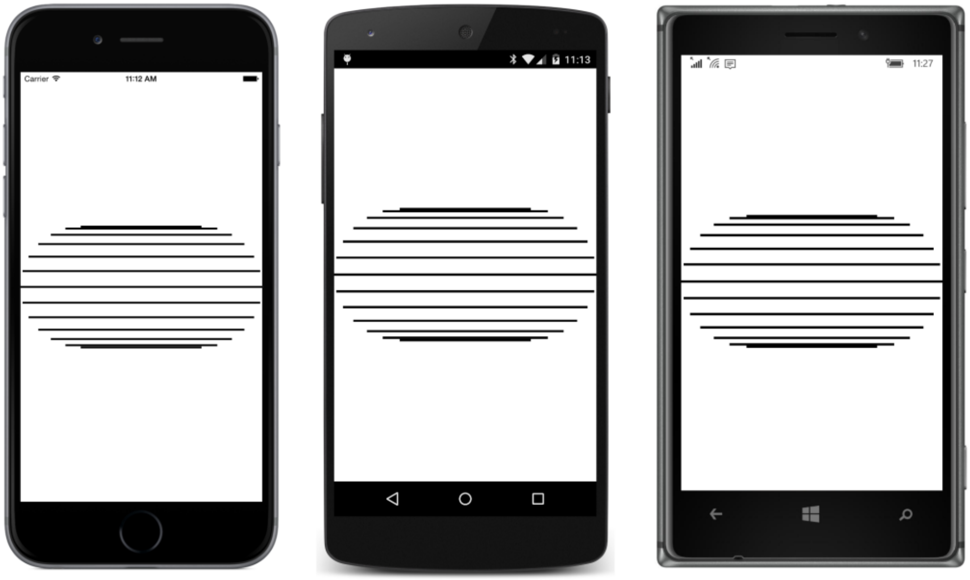
Everything else is done in code. The constructor adds 24 black `BoxView` elements to the `AbsoluteLayout`, and the `SizeChanged` handler for the page positions them in the spoke pattern:

```
public partial class RotatingSpokesPage : ContentPage
{
    const int numSpokes = 24;
    BoxView[] boxViews = new BoxView[numSpokes];

    public RotatingSpokesPage()
    {
        InitializeComponent();

        // Create all the BoxView elements.
        for (int i = 0; i < numSpokes; i++)
        {
            BoxView boxView = new BoxView
            {
                Color = Color.Black
            };
            boxViews[i] = boxView;
            absoluteLayout.Children.Add(boxView);
        }

        AnimationLoop();
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // Set AbsoluteLayout to a square dimension.
        double dimension = Math.Min(this.Width, this.Height);
        absoluteLayout.WidthRequest = dimension;
        absoluteLayout.HeightRequest = dimension;

        // Find the center and a size for the BoxView.
        Point center = new Point(dimension / 2, dimension / 2);
        Size boxViewSize = new Size(dimension / 2, 3);

        for (int i = 0; i < numSpokes; i++)
        {
            // Find an angle for each spoke.
            double degrees = i * 360 / numSpokes;
            double radians = Math.PI * degrees / 180;
```

```
            // Find the point of the center of each BoxView spoke.
            Point boxViewCenter =
                new Point(center.X + boxViewSize.Width / 2 * Math.Cos(radians),
                          center.Y + boxViewSize.Width / 2 * Math.Sin(radians));

            // Find the upper-left corner of the BoxView and position it.
            Point boxViewOrigin = boxViewCenter - boxViewSize * 0.5;
            AbsoluteLayout.SetLayoutBounds(boxViews[i],
                                        new Rectangle(boxViewOrigin, boxViewSize));

            // Rotate the BoxView around its center.
            boxViews[i].Rotation = degrees;
        }
    }
    …
}
```

Certainly the easiest way to render these spokes would be to position all 24 thin `BoxView` elements extending straight up from the center of the `AbsoluteLayout`—much like the initial 12:00 position of the hands of the `BoxViewClock` in the previous chapter—and then to rotate each of them around its bottom edge by an increment of 15 degrees. However, that requires that the `AnchorY` properties of these `BoxView` elements be set to 1 for that bottom edge rotation. That wouldn't work for this program because each of the `BoxView` elements must later be animated to rotate around its center.

The solution is to first calculate a position within the `AbsoluteLayout` for the *center* of each `BoxView`. This is the `Point` value in the `SizeChanged` handler called `boxViewCenter`. The `boxViewOrigin` is then the upper-left corner of the `BoxView` if the center of the `BoxView` is positioned at `boxViewCenter`. If you comment out the last statement in the `for` loop that sets the `Rotation` property of each `BoxView`, you'll see the spokes positioned like this:



All the horizontal lines (except for the top and bottom ones) are actually two aligned spokes. The center of each spoke is half the length of the spoke from the center of the page. Rotating each of the spokes around its center then creates the initial pattern you saw earlier.

Here's the `AnimationLoop` method:

```
public partial class RotatingSpokesPage : ContentPage
{
    …
    async void AnimationLoop()
    {
        // Keep still for 3 seconds.
        await Task.Delay(3000);

        // Rotate the configuration of spokes 3 times.
        uint count = 3;
        await absoluteLayout.RotateTo(360 * count, 3000 * count);

        // Prepare for creating Task objects.
        List<Task<bool>> taskList = new List<Task<bool>>(numSpokes + 1);

        while (true)
        {
            foreach (BoxView boxView in boxViews)
            {
                // Task to rotate each spoke.
                taskList.Add(boxView.RelRotateTo(360, 3000));
            }

            // Task to rotate the whole configuration.
            taskList.Add(absoluteLayout.RelRotateTo(360, 3000));
```

```
            // Run all the animations; continue in 3 seconds.
            await Task.WhenAll(taskList);

            // Clear the List.
            taskList.Clear();
        }
    }
}
```

After the preliminary rotation of only the `AbsoluteLayout` itself, the `while` block executes forever in rotating both the spokes and the `AbsoluteLayout`. Notice that a `List<Task<bool>>` is created for storing 25 simultaneous tasks. The `foreach` loop adds a `Task` to this `List` that calls `RelRotateTo` for each `BoxView` to rotate the spoke 360 degrees over three seconds. The final `Task` is another `RelRotateTo` on the `AbsoluteLayout` itself.

When using `RelRotateTo` in an animation that runs forever, the target `Rotation` property keeps getting larger and larger and larger. The actual rotation angle is the value of the `Rotation` property modulo 360.

Is the ever-increasing value of the `Rotation` property a potential problem?

In theory, no. Even if the underlying platform used a single-precision floating-point number to represent `Rotation` values, a problem wouldn't arise until the value exceeds 3.4×10^{38}. Even if you're increasing the `Rotation` property by 360 degrees every second, and you started the animation at the time of the Big Bang (13.8 billion years ago), the `Rotation` value would be only 4.4×10^{17}.

However, in reality, a problem can creep up, and much sooner than you might think. A `Rotation` angle of 36,000,000—just 100,000 rotations of 360 degrees—causes an object to be rendered a little differently than a `Rotation` angle of 0, and the deviation gets larger for higher `Rotation` angles.

If you'd like to explore this, you'll find a program named **RotationBreakdown** among the source code for this chapter. The program spins two `BoxView` elements at the same pace, one with `RotateTo` from 0 to 360 degrees, and the other with `RelRotateTo` with an argument of 36000. The `BoxView` rotated with `RotateTo` normally obscures the `BoxView` rotated with `RelRotateTo`, but that underlying `BoxView` is colored red, and within a minute you'll start seeing the red `BoxView` peek through. The deviation becomes greater the longer the program runs.

Often when you're combining animations, you want them all to start and end at the same time. But other times—and particularly with animations that run forever—you want several animations to run independently of each other, or at least seeming to run independently.

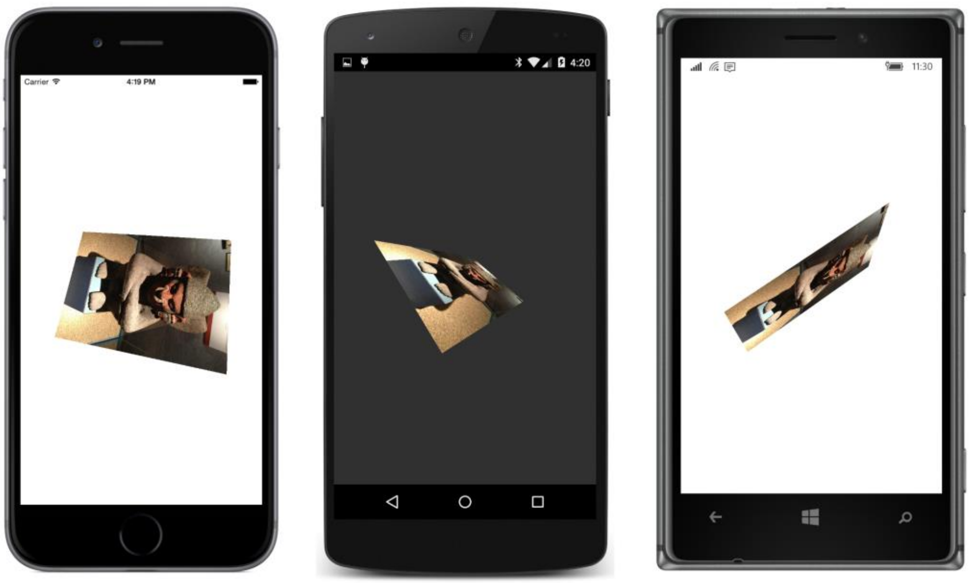
This is the case with the **SpinningImage** program. The program displays a bitmap using the `Image` element:

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

```
    <Image x:Name="image"
           Source="https://developer.xamarin.com/demo/IMG_0563.JPG"
           Scale="0.5" />

</ContentPage>
```

Normally, the `Image` would render the bitmap to fit within the screen while maintaining the bitmap's aspect ratio. In portrait mode, the width of the rendered bitmap would be the same as the width of the phone. However, with a `Scale` setting of 0.5, the `Image` is half that size.

The code-behind file then animates it by using `RotateTo`, `RotateXTo`, and `RotateYTo` to make it twist and turn almost randomly in space:



However, you probably don't want the `RotateTo`, `RotateXTo`, and `RotateYTo` to be synchronized in any way because that would result in repetitive patterns.

The solution here actually does create a repetitive pattern, but one that is five minutes in length. This is the duration for the three animations in the `Task.WhenAll` method:

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

        AnimationLoop();
    }

    async void AnimationLoop()
    {
```

```
        uint duration = 5 * 60 * 1000;  // 5 minutes

        while (true)
        {
            await Task.WhenAll(
                image.RotateTo(307 * 360, duration),
                image.RotateXTo(251 * 360, duration),
                image.RotateYTo(199 * 360, duration));

            image.Rotation = 0;
            image.RotationX = 0;
            image.RotationY = 0;
        }
    }
}
```

During this five-minute period, the three separate animations each makes a different number of 360 degree rotations: 307 rotations for `RotateTo`, 251 for `RotateXTo`, and 199 for `RotateYTo`. Those are all prime numbers. They have no common factors. So never during that five-minute period will any two of these rotations coincide with each other in the same way.

There's another way to create simultaneous but autonomous animations, but it requires going deeper into the animation system. That will be coming up soon.

Animating the Bounds property

Perhaps the most curious extension method in `ViewExtensions` class is `LayoutTo`. The argument is a `Rectangle` value, and the first question might be: What property is this method animating? The only property of type `Rectangle` defined by `VisualElement` is the `Bounds` property. This property indicates the position of an element relative to its parent and its size, but the property is get-only.

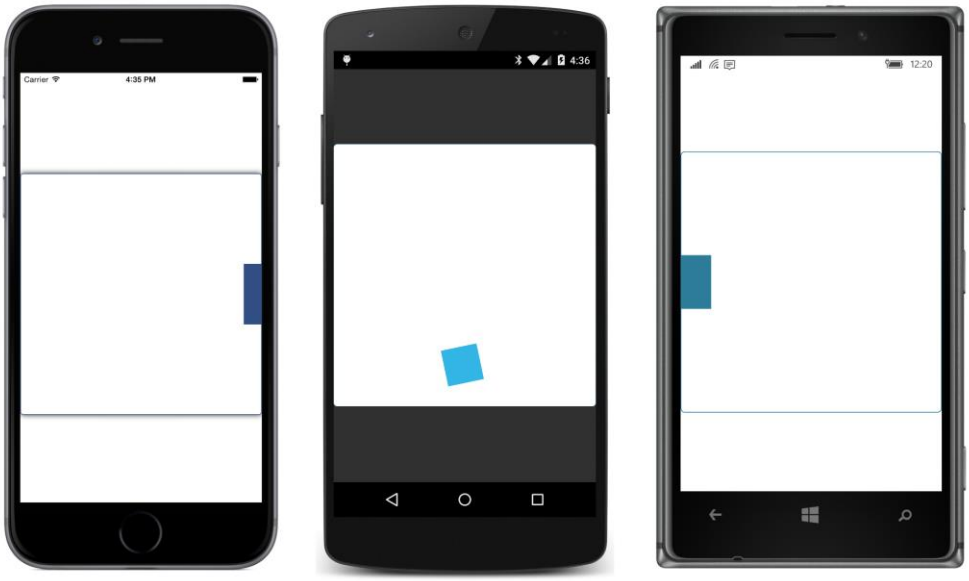
The `LayoutTo` animation does indeed animate the `Bounds` property, but it does so indirectly by calling the `Layout` method. The `Layout` method is not something that applications normally call. As the name suggests, it's commonly used within the layout system to position and size children relative to their parents. The only time you'll probably have an occasion to call `Layout` is when you write a custom layout class that derives from `Layout<View>`, as you'll see in Chapter 26, "Custom layouts."

You probably don't want to use the `LayoutTo` animation for children of a `StackLayout` or `Grid` because the animation overrides the position and size set by the parent. As soon as you turn the phone sideways, the page undergoes another layout pass that causes the `StackLayout` or `Grid` to move and size the child based on the normal layout process, and that will override your animation.

You'll have the same problem with a child of an `AbsoluteLayout`. After the `LayoutTo` animation completes, if you turn the phone sideways, the `AbsoluteLayout` then moves and sizes the child based on the child's `LayoutBounds` attached bindable property. But with `AbsoluteLayout` you also have a solution to this problem: After the `LayoutTo` animation concludes, the program can set the child's `LayoutBounds` attached bindable property to the same rectangle specified in the animation, perhaps using the final setting of the `Bounds` property set by the animation.

Keep in mind, however, that the `Layout` method and the `LayoutTo` animation have no knowledge of the proportional positioning and sizing feature in `AbsoluteLayout`. If you use proportional positioning and sizing, you might need to translate between proportional and absolute coordinates and sizes. The `Bounds` property always reports position and size in absolute coordinates.

The **BouncingBox** program uses `LayoutTo` to methodically bounce a `BoxView` around the interior of a square `Frame`. The `BoxView` starts at the center of the top edge, then moves in an arc to the center of the right edge, and then to the center of the bottom edge, the center of the left edge, and back up to the top, from where the journey continues. As the `BoxView` hits each edge, it realistically compresses and then expands like a rubber ball:



The code-behind file uses `AbsoluteLayout.SetLayoutBounds` to position the `BoxView` against each of the four edges, `LayoutTo` for the compression and decompression against the edge, and `RotateTo` to move the `BoxView` in an arc to the next edge.

The XAML file creates the `Frame`, the `AbsoluteLayout`, and the `BoxView`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BouncingBox.BouncingBoxPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentView SizeChanged="OnContentViewSizeChanged">
        <Frame x:Name="frame"
               OutlineColor="Accent"
               BackgroundColor="White"
```

```
               Padding="0"
               HorizontalOptions="Center"
               VerticalOptions="Center">
            <AbsoluteLayout SizeChanged="OnAbsoluteLayoutSizeChanged">
                <BoxView x:Name="boxView"
                         Color="Accent"
                         IsVisible="False" />
            </AbsoluteLayout>
        </Frame>
    </ContentView>
</ContentPage>
```

In the code-behind file, the `SizeChanged` handler for the `ContentView` adjusts the size of the `Frame` to be square, while the `SizeChanged` handler for the `AbsoluteLayout` saves its size for the animation calculations and starts the animation going if the size appears to be legitimate. (Without this check, the animation begins too early, and it uses an invalid size of the `AbsoluteLayout`.)

```
public partial class BouncingBoxPage : ContentPage
{
    static readonly uint arcDuration = 1000;
    static readonly uint bounceDuration = 250;
    static readonly double boxSize = 50;
    double layoutSize;
    bool animationGoing;

    public BouncingBoxPage()
    {
        InitializeComponent();
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
        double size = Math.Min(contentView.Width, contentView.Height);
        frame.WidthRequest = size;
        frame.HeightRequest = size;
    }

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        AbsoluteLayout absoluteLayout = (AbsoluteLayout)sender;
        layoutSize = Math.Min(absoluteLayout.Width, absoluteLayout.Height);

        // Only start the animation with a valid size.
        if (!animationGoing && layoutSize > 100)
        {
            animationGoing = true;
            AnimationLoop();
        }
    }
    …
}
```

The `AnimationLoop` method is lengthy, but that's only because it uses separate logic for each of the four sides and the transitions between those sides. For each side, the first step is to position the `BoxView` by using `AbsoluteLayout.SetLayoutBounds`. Then the `BoxView` is rotated in an arc to the next side. This requires setting the `AnchorX` and `AnchorY` properties so that the center of animation is close to the corner of the `Frame` but expressed in units of the `BoxView` size.

Then come the two calls to `LayoutTo` to animate the compression of the `BoxView` as it hits the inside of the `Frame`, and the subsequent expansion of `BoxView` as it bounces off:

```
public partial class BouncingBoxPage : ContentPage
{
    …
    async void AnimationLoop()
    {
        while (true)
        {
            // Initial position at top.
            AbsoluteLayout.SetLayoutBounds(boxView,
                new Rectangle((layoutSize - boxSize) / 2, 0, boxSize, boxSize));

            // Arc from top to right.
            boxView.AnchorX = layoutSize / 2 / boxSize;
            boxView.AnchorY = 0.5;
            await boxView.RotateTo(-90, arcDuration);

            // Bounce on right.
            Rectangle rectNormal = new Rectangle(layoutSize - boxSize,
                                                 (layoutSize - boxSize) / 2,
                                                 boxSize, boxSize);

            Rectangle rectSquashed = new Rectangle(rectNormal.X + boxSize / 2,
                                                   rectNormal.Y - boxSize / 2,
                                                   boxSize / 2, 2 * boxSize);

            boxView.BatchBegin();
            boxView.Rotation = 0;
            boxView.AnchorX = 0.5;
            boxView.AnchorY = 0.5;
            AbsoluteLayout.SetLayoutBounds(boxView, rectNormal);
            boxView.BatchCommit();

            await boxView.LayoutTo(rectSquashed, bounceDuration, Easing.SinOut);
            await boxView.LayoutTo(rectNormal, bounceDuration, Easing.SinIn);

            // Arc from right to bottom.
            boxView.AnchorX = 0.5;
            boxView.AnchorY = layoutSize / 2 / boxSize;
            await boxView.RotateTo(-90, arcDuration);

            // Bounce at bottom.
            rectNormal = new Rectangle((layoutSize - boxSize) / 2,
                                       layoutSize - boxSize,
                                       boxSize, boxSize);
```

```
rectSquashed = new Rectangle(rectNormal.X - boxSize / 2,
                             rectNormal.Y + boxSize / 2,
                             2 * boxSize, boxSize / 2);

boxView.BatchBegin();
boxView.Rotation = 0;
boxView.AnchorX = 0.5;
boxView.AnchorY = 0.5;
AbsoluteLayout.SetLayoutBounds(boxView, rectNormal);
boxView.BatchCommit();

await boxView.LayoutTo(rectSquashed, bounceDuration, Easing.SinOut);
await boxView.LayoutTo(rectNormal, bounceDuration, Easing.SinIn);

// Arc from bottom to left.
boxView.AnchorX = 1 - layoutSize / 2 / boxSize;
boxView.AnchorY = 0.5;
await boxView.RotateTo(-90, arcDuration);

// Bounce at left.
rectNormal = new Rectangle(0, (layoutSize - boxSize) / 2,
                           boxSize, boxSize);

rectSquashed = new Rectangle(rectNormal.X,
                             rectNormal.Y - boxSize / 2,
                             boxSize / 2, 2 * boxSize);

boxView.BatchBegin();
boxView.Rotation = 0;
boxView.AnchorX = 0.5;
boxView.AnchorY = 0.5;
AbsoluteLayout.SetLayoutBounds(boxView, rectNormal);
boxView.BatchCommit();

await boxView.LayoutTo(rectSquashed, bounceDuration, Easing.SinOut);
await boxView.LayoutTo(rectNormal, bounceDuration, Easing.SinIn);

// Arc from left to top.
boxView.AnchorX = 0.5;
boxView.AnchorY = 1 - layoutSize / 2 / boxSize;
await boxView.RotateTo(-90, arcDuration);

// Bounce on top.
rectNormal = new Rectangle((layoutSize - boxSize) / 2, 0,
                           boxSize, boxSize);

rectSquashed = new Rectangle(rectNormal.X - boxSize / 2, 0,
                             2 * boxSize, boxSize / 2);

boxView.BatchBegin();
boxView.Rotation = 0;
boxView.AnchorX = 0.5;
boxView.AnchorY = 0.5;
AbsoluteLayout.SetLayoutBounds(boxView, rectNormal);
```

```
            boxView.BatchCommit();

            await boxView.LayoutTo(rectSquashed, bounceDuration, Easing.SinOut);
            await boxView.LayoutTo(rectNormal, bounceDuration, Easing.SinIn);
        }
    }
}
```

The `SinOut` and `SinIn` easing functions provide a little realism for the compression to slow down as it's ending, and for the expansion to speed up after it's started.

Notice the calls to `BatchBegin` and `BatchCommit` that surround a number of property settings that accompany the positioning of the `BoxView` at one of the edges. These were added because there seemed to be a little flickering on the iPhone simulator, as if the properties were not being set simultaneously. However, the flickering remained even with these calls.

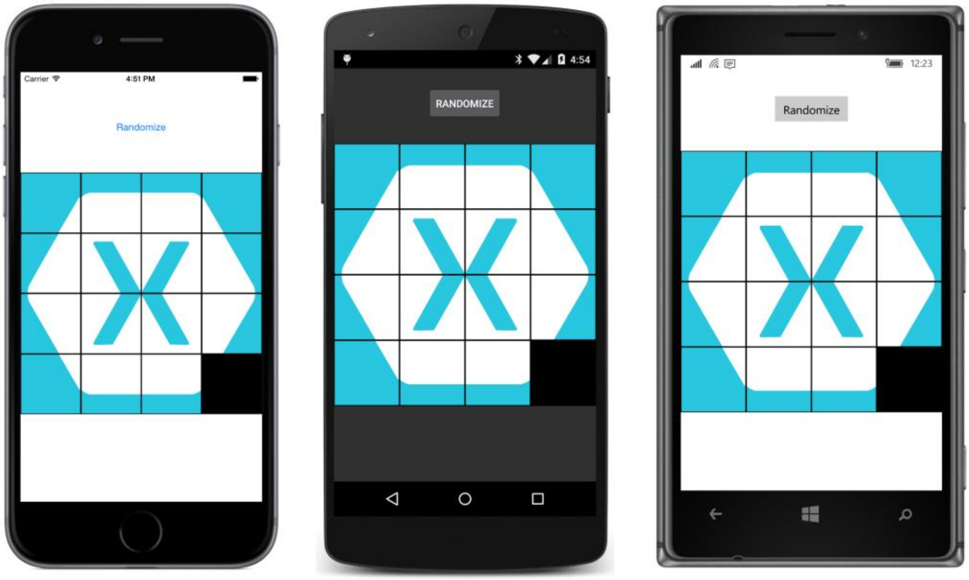
The `LayoutTo` animation is also used in one of the first games that was written for Xamarin.Forms. It's a version of the famous 15-Puzzle that consists of 15 tiles and one empty square in a four-by-four grid. The tiles can be shifted around but only by moving a tile into the empty spot.

On the early Apple Macintosh, this puzzle was named Puzzle. In the first Windows Software Development Kit, it was the only sample program using Microsoft Pascal, and it had the name Muzzle (for "Microsoft puzzle"). The version for Xamarin.Forms is thus called **Xuzzle**.

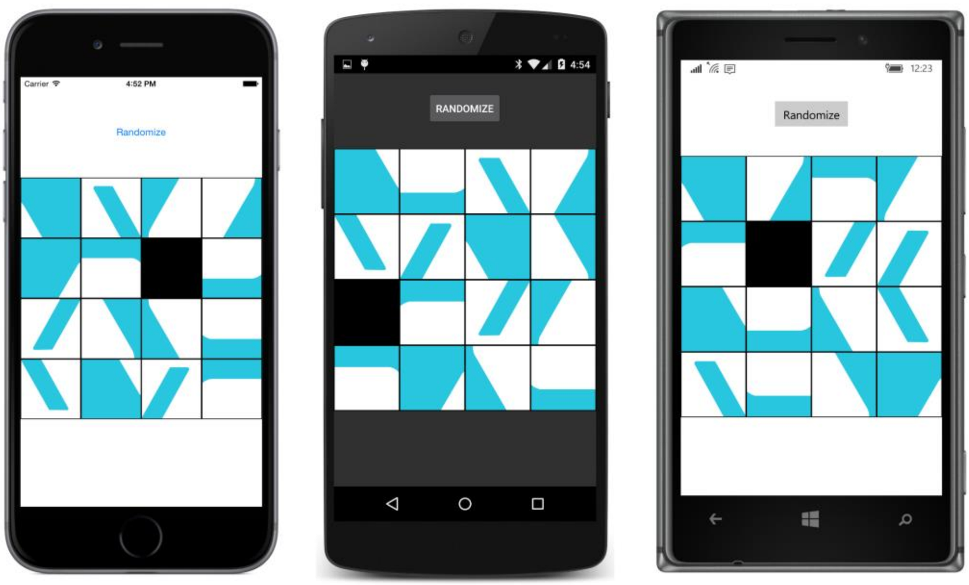
The original version of **Xuzzle** is here:

https://developer.xamarin.com/samples/xamarin-forms/Xuzzle/

The somewhat simplified version presented in this chapter doesn't include the animation that awards you for successfully completing the puzzle. However, rather than displaying letters or numbers, the tiles in this new version display 15/16 of the beloved Xamarin logo, called the Xamagon, and hence this new version is called **XamagonXuzzle**. Here's the startup screen:



When you press the **Randomize** button, the tiles are shifted around:



Your job is to shift the tiles back into their original configuration. You do this by tapping any tile adjacent to the empty square. The program applies an animation to shift the tapped tile into that empty square, and the empty square now replaces the tile you tapped.

You can also move multiple tiles with one tap. For example, suppose you tap the rightmost tile in the third row of the Android screen. The second tile in that row moves left, followed by the third and fourth tiles also moving left, again leaving the empty square replacing the tile you tapped.

The bitmaps for the 15 tiles were created especially for this program, and the **XamagonXuzzle** project contains them in the **Images** folder of the Portable Class Library, all with a **Build Action** of **Embedded Resource**.

Each tile is a `ContentView` that simply contains an `Image` with a little `Padding` applied for the gaps between the tiles that you see in the screenshots:

```
class XamagonXuzzleTile : ContentView
{
   public XamagonXuzzleTile (int row, int col, ImageSource imageSource)
   {
        Row = row;
        Col = col;

        Padding = new Thickness(1);
        Content = new Image
        {
            Source = imageSource
        };
   }

   public int Row { set; get; }

   public int Col { set; get; }
}
```

Each tile has an initial row and column, but the `Row` and `Col` properties are public, so the program can change them as the tiles are moved around. Also supplied to the constructor of the `XamagonXuzzleTile` class is an `ImageSource` object that references one of the bitmap resources.

The XAML file instantiates the `Button` and an `AbsoluteLayout` for the tiles:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamagonXuzzle.XamagonXuzzlePage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <ContentView SizeChanged="OnContentViewSizeChanged">
        <StackLayout x:Name="stackLayout">
            <Button Text="Randomize"
                    Clicked="OnRandomizeButtonClicked"
                    HorizontalOptions="CenterAndExpand"
                    VerticalOptions="CenterAndExpand" />

            <AbsoluteLayout x:Name="absoluteLayout"
                            BackgroundColor="Black" />

            <!-- Balance out layout with invisible button. -->
```

```
            <Button Text="Randomize"
                    Opacity="0"
                    HorizontalOptions="CenterAndExpand"
                    VerticalOptions="CenterAndExpand" />
        </StackLayout>
    </ContentView>
</ContentPage>
```

As you'll see, the `SizeChanged` handler for the `ContentView` changes the orientation of the `StackLayout` to accommodate portrait and landscape modes.

The constructor of the code-behind file instantiates all 15 tiles and gives each one an `ImageSource` based on one of the 15 bitmaps.

```
public partial class XamagonXuzzlePage : ContentPage
{
    // Number of tiles horizontally and vertically,
    //  but if you change it, some code will break.
    static readonly int NUM = 4;

    // Array of tiles, and empty row & column.
    XamagonXuzzleTile[,] tiles = new XamagonXuzzleTile[NUM, NUM];
    int emptyRow = NUM - 1;
    int emptyCol = NUM - 1;

    double tileSize;
    bool isBusy;

    public XamagonXuzzlePage()
    {
        InitializeComponent();

        // Loop through the rows and columns.
        for (int row = 0; row < NUM; row++)
        {
            for (int col = 0; col < NUM; col++)
            {
                // But skip the last one!
                if (row == NUM - 1 && col == NUM - 1)
                    break;

                // Get the bitmap for each tile and instantiate it.
                ImageSource imageSource =
                    ImageSource.FromResource("XamagonXuzzle.Images.Bitmap" +
                                             row + col + ".png");

                XamagonXuzzleTile tile = new XamagonXuzzleTile(row, col, imageSource);

                // Add tap recognition.
                TapGestureRecognizer tapGestureRecognizer = new TapGestureRecognizer
                {
                    Command = new Command(OnTileTapped),
                    CommandParameter = tile
                };
```

```
                tile.GestureRecognizers.Add(tapGestureRecognizer);

                // Add it to the array and the AbsoluteLayout.
                tiles[row, col] = tile;
                absoluteLayout.Children.Add(tile);
            }
        }
    }
    …
}
```

The `SizeChanged` handler for the `ContentView` has the responsibility of setting the `Orientation` property of the `StackLayout`, sizing the `AbsoluteLayout`, and sizing and positioning all the tiles within the `AbsoluteLayout`. Notice that each tile's position is calculated based on the `Row` and `Col` properties of that tile:

```
public partial class XamagonXuzzlePage : ContentPage
{
    …
    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
        double width = contentView.Width;
        double height = contentView.Height;

        if (width <= 0 || height <= 0)
            return;

        // Orient StackLayout based on portrait/landscape mode.
        stackLayout.Orientation = (width < height) ? StackOrientation.Vertical :
                                                     StackOrientation.Horizontal;

        // Calculate tile size and position based on ContentView size.
        tileSize = Math.Min(width, height) / NUM;
        absoluteLayout.WidthRequest = NUM * tileSize;
        absoluteLayout.HeightRequest = NUM * tileSize;

        foreach (View view in absoluteLayout.Children)
        {
            XamagonXuzzleTile tile = (XamagonXuzzleTile)view;

            // Set tile bounds.
            AbsoluteLayout.SetLayoutBounds(tile, new Rectangle(tile.Col * tileSize,
                                                               tile.Row * tileSize,
                                                               tileSize,
                                                               tileSize));
        }
    }
    …
}
```

The constructor has set a `TapGestureRecognizer` on each tile, and that's handled by the `OnTile-`

Tapped method. It's possible for a single tap to result in up to three tiles being shifted. That job is handled by the ShiftIntoEmpty method, which loops through all the shifted tiles and calls AnimateTile for each one. That method defines the Rectangle value for the call to LayoutTo—which is the one and only animation method in this entire program—and then other variables are adjusted for the new configuration:

```
public partial class XamagonXuzzlePage : ContentPage
{
    …
    async void OnTileTapped(object parameter)
    {
        if (isBusy)
            return;

        isBusy = true;
        XamagonXuzzleTile tappedTile = (XamagonXuzzleTile)parameter;
        await ShiftIntoEmpty(tappedTile.Row, tappedTile.Col);
        isBusy = false;
    }

    async Task ShiftIntoEmpty(int tappedRow, int tappedCol, uint length = 100)
    {
        // Shift columns.
        if (tappedRow == emptyRow && tappedCol != emptyCol)
        {
            int inc = Math.Sign(tappedCol - emptyCol);
            int begCol = emptyCol + inc;
            int endCol = tappedCol + inc;

            for (int col = begCol; col != endCol; col += inc)
            {
                await AnimateTile(emptyRow, col, emptyRow, emptyCol, length);
            }
        }
        // Shift rows.
        else if (tappedCol == emptyCol && tappedRow != emptyRow)
        {
            int inc = Math.Sign(tappedRow - emptyRow);
            int begRow = emptyRow + inc;
            int endRow = tappedRow + inc;

            for (int row = begRow; row != endRow; row += inc)
            {
                await AnimateTile(row, emptyCol, emptyRow, emptyCol, length);
            }
        }
    }

    async Task AnimateTile(int row, int col, int newRow, int newCol, uint length)
    {
        // The tile to be animated.
        XamagonXuzzleTile animaTile = tiles[row, col];
```

```
        // The destination rectangle.
        Rectangle rect = new Rectangle(emptyCol * tileSize,
                                       emptyRow * tileSize,
                                       tileSize,
                                       tileSize);

        // Animate it!
        await animaTile.LayoutTo(rect, length);

        // Set layout bounds to same Rectangle.
        AbsoluteLayout.SetLayoutBounds(animaTile, rect);

        // Set several variables and properties for new layout.
        tiles[newRow, newCol] = animaTile;
        animaTile.Row = newRow;
        animaTile.Col = newCol;
        tiles[row, col] = null;
        emptyRow = row;
        emptyCol = col;
    }
    …
}
```

The `AnimateTile` method uses `await` for the `LayoutTo` call. If it did not use `await`—if it let the `LayoutTo` animation run in the background while it proceeded with its other work—then the program would not know when the `LayoutTo` animation concluded. That means that if `ShiftIntoEmpty` were shifting two or three tiles, those animations would occur simultaneously instead of sequentially.

Because `AnimateTile` uses `await`, the method must have the `async` modifier. However, if the method returned `void`, then the `AnimateTile` method would return when the `LayoutTo` animation begins, and again the `ShiftIntoEmpty` method would not know when the animation completes. For this reason, `AnimateTile` returns a `Task` object. The `AnimateTile` method still returns when the `LayoutTo` animation begins, but it returns a `Task` object that can signal when the `AnimateTile` method completes. This means that `ShiftIntoEmpty` can call `AnimateTile` using `await` and move the tiles sequentially.

`ShiftIntoEmpty` uses `await`, so it must also be defined with the `async` modifier, but it could return `void`. If so, then `ShiftIntoEmpty` would return at the time it makes its first call to `AnimateTile`, which means that the `OnTileTapped` method would not know when the entire animation has completed. But `OnTileTapped` needs to prevent tiles from being tapped and animated if they are already in the process of being animated, which requires that `ShiftIntoEmpty` return `Task`. This means that `OnTileTapped` can use `await` with `ShiftIntoEmpty`, which means that `OnTileTapped` must also include the `async` modifier.

The `OnTileTapped` handler is called from the `Button` itself, so it cannot return `Task`. It must return `void`, just as the method is defined. But you can see how the use of `await` and `async` seems to ripple up the chain of method calls.

Once the code exists for handling taps, implementing the **Randomize** button becomes fairly trivial.

It simply makes multiple calls to `ShiftIntoEmpty` with a faster animation speed:

```
public partial class XamagonXuzzlePage : ContentPage
{
    …
    async void OnRandomizeButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        button.IsEnabled = false;
        Random rand = new Random();
        isBusy = true;

        // Simulate some fast crazy taps.
        for (int i = 0; i < 100; i++)
        {
            await ShiftIntoEmpty(rand.Next(NUM), emptyCol, 25);
            await ShiftIntoEmpty(emptyRow, rand.Next(NUM), 25);
        }
        button.IsEnabled = true;
        isBusy = false;
    }
}
```

Again, using `await` with the `ShiftIntoEmpty` calls allows the calls to be executed sequentially (which is exciting to watch) and allows the `OnRandomizeButtonClicked` handler to know when everything is completed so it can reenable the `Button` and allow taps on the tiles.

Your own awaitable animations

In the next section of this chapter, you'll see the underlying animation infrastructure that Xamarin.Forms implements. These underlying methods allow you to define your own animation functions that return `Task` objects and which can be used with `await`.

In Chapter 20, "Async and file I/O," you saw how to use the static `Task.Run` method to create a secondary thread of execution for carrying out an intensive background job like a Mandelbrot computation. The `Task.Run` method returns a `Task` object that can signal when the background job has completed.

But animation is not quite like that. An animation doesn't need to spend a lot of time crunching numbers. It merely needs to do something very brief and simple—such as setting a `Rotation` property—once every 16 milliseconds. That job can run in the user-interface thread—in fact, the actual property access *must* run in the user-interface thread—and the timing can be handled by using `Device.StartTimer` or `Task.Delay`.

You shouldn't use `Task.Run` for implementing animations, because a secondary thread of execution is unnecessary and wasteful. However, when you actually sit down to write an animation method similar to the Xamarin.Forms animation methods such as `RotateTo`, you might encounter an obstacle. The method must return a `Task` object and perhaps use `Device.StartTimer` for the timing, but that doesn't seem possible.

Here's a first stab at writing such a method. The parameters include the target `VisualElement`, *from* and *to* values, and a duration. It uses `Device.StartTimer` and a `Stopwatch` to calculate the current setting of the `Rotation` property, and it exits the `Device.StartTimer` callback when the animation has completed:

```
Task MyRotate(VisualElement visual, double fromValue, double toValue, uint duration)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    Device.StartTimer(TimeSpan.FromMilliseconds(16), () =>
        {
            double t = Math.Min(1, stopwatch.ElapsedMilliseconds / (double)duration);
            double value = fromValue + t * (toValue - fromValue);
            visual.Rotation = value;
            bool completed = t == 1;

            if (completed)
            {
                // Need to signal that the Task has completed. But how?
            }
            return !completed;
        });

    // Need to return a Task object here but where does it come from?
}
```

At two crucial points the method doesn't know what to do. After the method calls `Device.StartTimer`, it needs to exit and return a `Task` object to the caller. But where does this `Task` object come from? The `Task` class has a constructor, but like `Task.Run`, that constructor creates a second thread of execution, and there's no reason to create that thread. Moreover, when the animation has finished, the method somehow needs to signal that the `Task` has completed.

Fortunately, there exists a class that does exactly what you want. It's called `TaskCreationSource`. It's a generic class in which the type parameter is the same as the type parameter of the `Task` object that you want to create. The `Task` property of the `TaskCreationSource` object provides the `Task` object you need. This is what your asynchronous method returns. When your method has completed processing the background job, it can call `SetResult` on the `TaskCreationSource` object, signaling that the job is finished.

The following **TryAwaitableAnimation** program shows how to use `TaskCreationSource` in a `MyRotateTo` method that is called from the `Clicked` handler of a `Button`:

```
public partial class TryAwaitableAnimationPage : ContentPage
{
    public TryAwaitableAnimationPage()
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
```

```
    {
        Button button = (Button)sender;
        uint milliseconds = UInt32.Parse((string)button.StyleId);
        await MyRotate(button, 0, 360, milliseconds);
    }

    Task MyRotate(VisualElement visual, double fromValue, double toValue, uint duration)
    {
        TaskCompletionSource<object> taskCompletionSource = new TaskCompletionSource<object>();

        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        Device.StartTimer(TimeSpan.FromMilliseconds(16), () =>
            {
                double t = Math.Min(1, stopwatch.ElapsedMilliseconds / (double)duration);
                double value = fromValue + t * (toValue - fromValue);
                visual.Rotation = value;
                bool completed = t == 1;

                if (completed)
                {
                    taskCompletionSource.SetResult(null);
                }
                return !completed;
            });

        return taskCompletionSource.Task;
    }
}
```

Notice the instantiation of `TaskCreationSource`, the return value of the `Task` property of that object, and the call to `SetResult` within the `Device.StartTimer` callback when the animation has finished.

There is no nongeneric form of `TaskCreationSource`, so if your method just returns a `Task` object rather than a `Task<T>` object, you'll need to specify a type when defining the `TaskCreationSource` instance. By convention, you can use `object` for this purpose, in which case your method calls `SetResult` with a `null` argument.

The **TryAwaitableAnimation** XAML file instantiates three `Button` elements that share this `Clicked` handler. Each of them defines its own animation duration as the `StyleId` property. (As you'll recall, `StyleId` is not used within Xamarin.Forms and exists solely to be used by an application programmer as a convenient way to attach arbitrary data to an element.)

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TryAwaitableAnimation.TryAwaitableAnimationPage">
    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="Button">
```

```
                <Setter Property="Text" Value="Tap Me!" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </StackLayout.Resources>

    <Button Clicked="OnButtonClicked" StyleId="5000" />

    <Button Clicked="OnButtonClicked" StyleId="2500" />

    <Button Clicked="OnButtonClicked" StyleId="1000" />
  </StackLayout>
</ContentPage>
```

Even though each of these `Button` elements is animating itself by a call to `MyRotate`, you can have all buttons spinning at the same time. Each call to `MyRotate` gets its own set of local variables, and these local variables are used in each `Device.StartTimer` callback.

However, if you tap a `Button` while it's still spinning, then a second animation is applied to that `Button` and the two animations battle each other. What the code requires is a way to cancel the previous animation when a new animation is applied.

One approach is for the `MyRotate` method to maintain a dictionary of type `Dictionary<VisualElement,bool>` defined as a field. Whenever it begins an animation, `MyRotate` adds the target `VisualElement` as a key to this dictionary with a value of `false`. When the animation ends, it removes this entry from the dictionary. A separate method (named `CancelMyRotate`, perhaps) can set the value in the dictionary to `true`, meaning to cancel the animation. The `Device.StartTimer` callback can begin by checking the value of the dictionary for the particular `VisualElement` and return `false` from the callback if the animation has been cancelled. But you'll discover in the discussion that follows how to do it with less code.

Now that you've seen the high-level animation functions implemented in the `ViewExtensions` class, let's explore how the rest of the Xamarin.Forms animation system implements these functions and allows you to start, control, and cancel animations.

Deeper into animation

On first encounter, the complete Xamarin.Forms animation system can be a little confusing. Let's begin with a global view of the three public classes that you can use to define animations.

Sorting out the classes

In addition to the `Easing` class, the Xamarin.Forms animation system comprises three public classes. Here they are in hierarchical order from high level to low level:

ViewExtensions class

This is the class you've already seen. `ViewExtensions` is a static class that contains several extension methods for `VisualElement`, which is the parent class to `View` and `Page`:

- `TranslateTo` animates the `TranslationX` and `TranslationY` properties
- `ScaleTo` animates the `Scale` property
- `RelScaleTo` applies an animated incremental increase or decrease to the `Scale` property
- `RotateTo` animates the `Rotation` property
- `RelRotateTo` applies an animated incremental increase or decrease to the `Rotation` property
- `RotateXTo` animates the `RotationX` property
- `RotateYTo` animates the `RotationY` property
- `FadeTo` animates the `Opacity` property
- `LayoutTo` animates the get-only `Bounds` property by calling the `Layout` method

As you can see, the first seven methods target transform properties. These properties do not cause any change to how the element is perceived in layout. Although the animated view can move, change size, and rotate, none of the other views on the page are affected, except possibly being obscured by the new location or size.

The `FadeTo` animation changes only the `Opacity` property, so that doesn't cause layout changes either.

As you've seen, the `LayoutTo` animation is a little different. The argument is a `Rectangle` value, and the method essentially overrides the location and size assigned to the view by the element's parent `Layout` or `Layout<T>` object. `LayoutTo` is most useful for animating children of an `AbsoluteLayout` because you can call `AbsoluteLayout.SetLayoutBounds` with the same `Rectangle` object after the animation has completed. In Chapter 26, you'll learn how to use `LayoutTo` in a class that derives from `Layout<View>`.

These are all asynchronous methods that return `Task<bool>`. The Boolean return value is `true` if the animation was cancelled and `false` if it ran to completion.

In addition, `ViewExtensions` also contains a static `ViewExtensions.CancelAnimations` method (not an extension method) that has a single argument of type `VisualElement`. This method cancels any and all animations started with this class on that `VisualElement` object.

All the extension methods in `ViewExtensions` work by creating one or more `Animation` objects and then calling the `Commit` method defined by that `Animation` class.

The Animation class

The `Animation` class has two constructors: a parameterless constructor and another with five parameters, although only one of the arguments is required:

```
public Animation (Action<double> callback,
                  double start = 0.0f,
                  double end = 1.0f,
                  Easing easing = null,
                  Action finished = null)
```

This defines an animation of a `double` value that begins at `start` and ends at `end`. Often, these two arguments will have their default values of 0 and 1, respectively. The animated value is passed to the callback method as an argument, where it is generally named `t` or `progress`. The callback can do whatever it wants with this value, but generally it's used to change a value of a property. If the target property is of type `double`, then `start` and `end` values can define the start and end values of the animated property directly.

`Animation` implements the `IEnumerable` interface. It can maintain a collection of child animations that can then be uniformly started and remain synchronized. To allow a program to add items to this collection, `Animation` defines four methods:

- `Add`
- `Insert`
- `WithConcurrent` (two versions)

These are all fundamentally the same in that they all add a child `Animation` object to an internal collection maintained by `Animation`. You'll see examples shortly.

Starting the animation (which might or might not include child animations) requires a call to the `Commit` method. The `Commit` method specifies the duration of the animation and also includes two more callbacks:

```
animation.Commit(IAnimatable owner,
                 string name,
                 uint rate = 16,
                 uint length = 250,
                 Easing easing = null,
                 Action<double, bool> finished = null,
                 Func<bool> repeat = null);
```

Notice the first argument is `IAnimatable`. The `IAnimatable` interface defines just two methods, named `BatchBegin` and `BatchCommit`. The only class that implements `IAnimatable` is `VisualElement`, which is the class associated with the `ViewExtensions` methods.

The `name` argument identifies the animation. You can use methods in the `AnimationExtensions` class (coming up) to determine if an animation of that name is running or to cancel it. You don't need to use unique names for every animation that you're running, but if you're making multiple overlapping Commit calls on the same visual object, then those names should be unique.

In theory, the rate argument indicates the number of milliseconds between each call to the callback method defined in the Animation constructor. It is set at 16 for an animation speed of 60 frames per second, but changing it has no effect.

The repeat callback allows the animation to be repeated. It's called at the end of the animation, and if the callback returns true, that signals that the animation should be repeated. As you'll see, it works in some configurations but not others.

The Commit method in the Animation class works by calling an Animate method in the AnimationExtensions class.

AnimationExtensions class

Like ViewExtensions, AnimationExtentions is a static class containing mostly extension methods. But while the first parameter in the ViewExtensions methods is a VisualElement, the first parameter in the AnimationExtensions methods is an IAnimatable to be consistent with the Commit method in the Animation class.

AnimationExtensions defines several overloads of the Animate method with callbacks and other information. The most extensive version of Animate is this generic method:

```
public static void Animate<T>(this IAnimatable self,
                              string name,
                              Func<double, T> transform,
                              Action<T> callback,
                              uint rate = 16,
                              uint length = 250,
                              Easing easing = null,
                              Action<T, bool> finished = null,
                              Func<bool> repeat = null);
```

In one sense, this is the only animation method you need. By now many of these parameters should be recognizable. But notice the transform method that can help structure the logic of animations that target properties that are not of type double.

For example, suppose you want to animate a property of type Color. You first write a little transform method that accepts a double argument ranging from 0 to 1 (and often named t or progress) and returns a Color value corresponding to that value. The callback method obtains that Color value and can then set it to a particular property of a particular object. You'll see this precise application at the end of this chapter.

Other public methods in the AnimationExtensions class are AnimationIsRunning to determine if a particular animation on a particular VisualElement instance is running, and AbortAnimation to cancel an animation. Both are extension methods for IAnimatable and require a name consistent with the name passed to the Animate method or the Commit method of Animation.

Working with the Animation class

Let's experiment a bit with the `Animation` class. This involves instantiating objects of type `Animation` and then calling `Commit`, which actually starts the animation going. The `Commit` method does not return a `Task` object; instead, the `Animation` class provides notifications entirely through callbacks.

There are several different ways to configure an `Animation` object, and some of these might involve child animations, which is why the project that demonstrates the `Animation` class is called **ConcurrentAnimations**. But not all the demonstrations in this program involve child animations.

The XAML file defines mostly a bunch of buttons that serve both to trigger animations and to be the targets of these animations:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ConcurrentAnimations.ConcurrentAnimationsPage">
    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="Button">
                    <Setter Property="HorizontalOptions" Value="Center" />
                    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>

        <Button Text="Animation 1 (Scale)"
                Clicked="OnButton1Clicked" />

        <Button Text="Animation 2 (Repeated)"
                Clicked="OnButton2Clicked" />

        <Button Text="Stop Animation 2"
                Clicked="OnStop2Clicked" />

        <Button Text="Animation 3 (Scale up &amp; down)"
                Clicked="OnButton3Clicked" />

        <Button Text="Animation 4 (Scale &amp; Rotate)"
                Clicked="OnButton4Clicked" />

        <Button Text="Animation 5 (Dots)"
                Clicked="OnButton5Clicked" />

        <Label x:Name="waitLabel"
               FontSize="Large"
               WidthRequest="100" />

        <Button Text="Turn off dots"
                Clicked="OnTurnOffButtonClicked" />

        <Button Text="Animation 6 (Color)"
                Clicked="OnButton6Clicked" />
```

```
    </StackLayout>
</ContentPage>
```

The code-behind file contains the event handlers for each of these buttons.

The code in the `Clicked` handler for the first `Button` uses comments to identify all the arguments for the `Animation` constructor and the `Commit` call. There are a total of four callback methods, each of which are expressed here as a lambda function but not with the most concise syntax:

```
public partial class ConcurrentAnimationsPage : ContentPage
{
    …
    public ConcurrentAnimationsPage()
    {
        InitializeComponent();
    }

    void OnButton1Clicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        Animation animation = new Animation(
            (double value) =>
                {
                    button.Scale = value;
                },              // callback
            1,                  // start
            5,                  // end
            Easing.Linear,      // easing
            () =>
                {
                    Debug.WriteLine("finished");
                }               // finished (but doesn't fire in this configuration)
            );

        animation.Commit(
            this,               // owner
            "Animation1",       // name
            16,                 // rate (but has no effect here)
            1000,               // length (in milliseconds)
            Easing.Linear,
            (double finalValue, bool wasCancelled) =>
                {
                    Debug.WriteLine("finished: {0} {1}", finalValue, wasCancelled);
                    button.Scale = 1;
                },              // finished
            () =>
                {
                    Debug.WriteLine("repeat");
                    return false;
                }                // repeat
            );
    }
```

```
    …
}
```

The callback in the `Animation` constructor sets the `Scale` property of the `Button` to the value passed to that callback. This value ranges from 1 to 5 as the next two arguments indicate.

The `Commit` method assigns an owner to the animation. This can be the visual element on which the animation is applied or another visual element, such as the page. The `name` is combined with the owner to uniquely identify the animation if it must be cancelled. The same owner should be used for calls to `AnimationIsRunning` or `AbortAnimation` in the `AnimationExtensions` class. (You'll see how to cancel an animation shortly.)

The last argument to the `Animation` constructor is named `finished`, and it's a callback that is supposed to be invoked when the animation completes, but in this configuration it is not called. Fortunately, the `Commit` method also has a `finished` callback with two arguments. The first should indicate a final value (but in this configuration that value is always 1), and the second argument is a `bool` that is set to `true` if the animation was cancelled.

In this example, both `finished` callbacks make calls to `Debug.WriteLine` so that you can confirm that one is called but not the other. The `finished` callback included with the `Commit` call sets the `Scale` property back to 1, so the `Button` snaps back to its original size.

If you want to apply an easing function, you can specify it either in the constructor or in the `Commit` method call.

The `Clicked` handler for the second `Button` is very similar to the first except that the syntax is considerably more concise. Many of the parameters to the constructor and the `Commit` method have default values, and the constructor has taken advantage of those. The syntax for the lambda functions has also been simplified:

```
public partial class ConcurrentAnimationsPage : ContentPage
{
    …
    void OnButton2Clicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        Animation animation = new Animation(v => button.Scale = v, 1, 5);
        animation.Commit(this, "Animation2", 16, 1000, Easing.Linear,
                         (v, c) => button.Scale = 1,
                         () => true);
    }

    void OnStop2Clicked(object sender, EventArgs args)
    {
        this.AbortAnimation("Animation2");
    }
    …
}
```

The only functional difference between the code for this `Button` and the previous `Button` involves the `repeat` callback. When the animation completes—that is, after a value of 5 is passed to the `callback` method—both the `repeat` and `finished` callbacks passed to the `Commit` method are called. If `repeat` returns `true`, then the animation starts over from the beginning, and at the end of that, `repeat` and `finished` are called again.

Fortunately, the XAML file includes another `Button` that calls `AbortAnimation` to terminate the animation. `AbortAnimation` is an extension method, so it must be called on the same element passed as the first argument to the `Commit` method, which in this case is the page object.

If you want several concurrent forever animations that run independently of each other, you can create an `Animation` object for each of them and then call `Commit` on each one with a `repeat` callback that returns `true`.

Child animations

Those first two examples in **ConcurrentAnimations** are single animations. The `Animation` class also supports child animations, and that's what the handler for the `Button` labeled "Animation 3" demonstrates. It first creates a parent `Animation` object with the parameterless constructor. It then creates two additional `Animation` objects and adds them to the parent `Animation` object with the `Add` and `Insert` methods:

```
public partial class ConcurrentAnimationsPage : ContentPage
{
    …
    void OnButton3Clicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        // Create parent animation object.
        Animation parentAnimation = new Animation();

        // Create "up" animation and add to parent.
        Animation upAnimation = new Animation(
            v => button.Scale = v,
            1, 5, Easing.SpringIn,
            () => Debug.WriteLine("up finished"));

        parentAnimation.Add(0, 0.5, upAnimation);

        // Create "down" animation and add to parent.
        Animation downAnimation = new Animation(
            v => button.Scale = v,
            5, 1, Easing.SpringOut,
            () => Debug.WriteLine("down finished"));

        parentAnimation.Insert(0.5, 1, downAnimation);

        // Commit parent animation.
        parentAnimation.Commit(
```

```
            this, "Animation3", 16, 5000, null,
            (v, c) => Debug.WriteLine("parent finished: {0} {1}", v, c));
    }
    …
}
```

These `Add` and `Insert` methods are basically the same, and in practical use are interchangeable. The only difference is that `Insert` returns the parent `Animation` object while `Add` does not.

Both methods require two arguments of type `double` with the names `beginAt` and `finishAt`. These two arguments must be between 0 and 1, and `finishAt` must be greater than `beginAt`. These two arguments indicate the relative period within the total animation that these particular child animations are active.

The total animation is five seconds long. That's the argument of 5000 in the `Commit` method. The first child animation animates the `Scale` property from 1 to 5. The `beginAt` and `finishAt` arguments are 0 and 0.5, respectively, which means that this child animation is active during the first half of the overall animation—that is, during the first 2.5 seconds. The second child animation takes the `Scale` property from 5 back down to 1. The `beginAt` and `finishAt` arguments are 0.5 and 1, respectively, which means that this animation occurs in the second half of the overall five-second animation.

The result is that the `Button` is scaled to five times its size over 2.5 seconds and then scaled back down to 1 over the final 2.5 seconds. But notice the two `Easing` functions set on the two child animations. The `Easing.SpringIn` object causes the `Button` to initially shrink in size before getting larger, and the `Easing.SpringOut` function also causes the `Button` to become smaller than its actual size toward the end of the complete animation.

As you'll see when you click the button to run this code, all the `finished` callbacks are now called. That is one difference between using the `Animation` class for a single animation and using it with child animations. The `finished` callback on the child animations indicates when that particular child has completed, and the `finished` callback passed to the `Commit` method indicates when the entire animation has finished.

There are two more differences when using child animations:

- When using child animations, returning `true` from the `repeat` callback on the `Commit` method doesn't cause the animation to repeat, but the animation will nevertheless continue to run with no new values.
- If you include an `Easing` function in the `Commit` method, and the `Easing` function returns a value greater than 1, the animation will be terminated at that point. If the `Easing` function returns a value less than 0, the value is clamped to equal 0.

If you want to use an `Easing` function that returns a value less than 0 or greater than 1 (for example, the `Easing.SpringIn` or `Easing.SpringOut` function), specify it in one or more of the child animations, as the example demonstrates, rather than the `Commit` method.

The C# compiler recognizes the `Add` method of a class that implements `IEnumerable` as a collection initializer. To keep the animation syntax to a minimum, you can follow the `new` operator on the parent `Animation` object with a pair of curly braces to initialize the contents with children. Each pair of curly braces within those outer curly braces encloses the arguments to the `Add` method. Here is an animation with three children:

```
public partial class ConcurrentAnimationsPage : ContentPage
{
    …
    void OnButton4Clicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;

        new Animation
        {
            { 0, 0.5, new Animation(v => button.Scale = v, 1, 5) },
            { 0.25, 0.75, new Animation(v => button.Rotation = v, 0, 360) },
            { 0.5, 1, new Animation(v => button.Scale = v, 5, 1) }
        }.Commit(this, "Animation4", 16, 5000);
    }
    …
}
```

Notice also that `Commit` is called directly on the `Animation` constructor. This is as concise as you can make this code.

The first two arguments to these implicit `Add` methods indicate where within the entire parent animation the child is active. The first child animates the `Scale` property and is active during the first half of the parent animation, and the last child also animates the `Scale` property and is active for the last half of the parent animation. That's the same as the previous example. But now there's also an animation of the `Rotation` property with start and end values of 0.25 and 0.75. This `Rotation` animation begins halfway through the first `Scale` animation and ends halfway through the second `Scale` animation. This is how child animations can be overlapped.

The `Animation` class also includes two methods named `WithConcurrent` to add child animations to a parent `Animation` object. These are similar to the `Add` and `Insert` methods, except that the `beginAt` and `finishAt` arguments (or `start` and `end` as they're called in one of the `WithConcurrent` methods) are not restricted to the range of 0 through 1. However, only that part of the child animation that corresponds to a range of 0 through 1 will be active.

For example, suppose you call `WithConcurrent` to define a child animation that targets a `Scale` property from 1 to 4, but with a `beginAt` argument of –1 and a `finishAt` argument of 2. The `beginAt` value of –1 corresponds to a `Scale` value of 1, and the `finishAt` value of 2 corresponds to a `Scale` value of 4, but values outside the range of 0 and 1 don't play a role in the animation, so the `Scale` property will only be animated from 2 to 3.

Beyond the high-level animation methods

The examples in **ConcurrentAnimations** that you've seen so far have restricted themselves to animations of the `Scale` and `Rotate` properties, so they haven't shown anything you can't do with the methods in the `ViewExtensions` class. But because you have access to the actual callback method, you can do anything you want during that callback.

Here's an animation that you might use to indicate that your application is performing an operation that might take some time to complete. Rather than displaying an `ActivityIndicator`, you've chosen to display a string of periods that repetitively increases in length from 0 to 10. Those two values are specified as arguments to the `Animation` constructor. The callback method casts the current value to an integer for use with one of the lesser-known `string` constructors to construct a string with that number of dots:

```
public partial class ConcurrentAnimationsPage : ContentPage
{
    bool keepAnimation5Running = false;
    …
    void OnButton5Clicked(object sender, EventArgs args)
    {
        Animation animation =
                    new Animation(v => dotLabel.Text = new string('.', (int)v), 0, 10);
        animation.Commit(this, "Animation5", 16, 3000, null,
                         (v, cancelled) => dotLabel.Text = "",
                         () => keepAnimation5Running);
        keepAnimation5Running = true;
    }

    void OnTurnOffButtonClicked(object sender, EventArgs args)
    {
        keepAnimation5Running = false;
    }
    …
}
```

The `OnButton5Clicked` method concludes by setting the `keepAnimation5Running` field to `true`, and the `repeat` callback in the `Commit` method returns that value. The animation will keep running until `keepAnimation5Running` is set to `false`, which is what the next `Button` does.

The difference between this technique and cancelling the animation is that this technique does not immediately end the animation. The `repeat` callback is only called after the animation reaches its `end` value (which is 10 in this case), so the animation could continue to run for almost another three seconds after `keepAnimation5Running` is set to `false`.

The final example in the **ConcurrentAnimations** program animates the `BackgroundColor` property of the page by setting it to `Color` values created by the `Color.FromHsla` method with hue values ranging from 0 through 1. This animation gives the effect of sweeping through the colors of the rainbow:

```
public partial class ConcurrentAnimationsPage : ContentPage
```

```
{
    …
    void OnButton6Clicked(object sender, EventArgs args)
    {
        new Animation(callback: v => BackgroundColor = Color.FromHsla(v, 1, 0.5),
                      start: 0,
                      end: 1).Commit(owner: this,
                                     name: "Animation6",
                                     length: 5000,
                                     finished: (v, c) => BackgroundColor = Color.Default);
    }
}
```

This code uses named arguments and hence illustrates yet another syntax variation for instantiating an `Animation` object and calling `Commit` on it.

More of your own awaitable methods

Earlier, you saw how to use `TaskCompletionSource` together with `Device.StartTimer` to write your own asynchronous animation methods. You can also combine `TaskCompletionSource` with the `Animation` class to write you own asynchronous animation methods similar to those in the `ViewExtensions` class.

Suppose you like the idea of the **SlidingEntrance** program, but you are dissatisfied that the `Easing.SpringOut` function doesn't work with the `TranslateTo` method. You can write your own translation animation method. If you only need to animate the `TranslationX` property, you can call it `TranslateXTo`:

```
public static Task<bool> TranslateXTo(this VisualElement view, double x,
                                      uint length = 250, Easing easing = null)
{
    easing = easing ?? Easing.Linear;
    TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();

    Animation animation = new Animation(
        (value) => view.TranslationX = value, // callback
        view.TranslationX,   // start
        x,                   // end
        easing);             // easing

    animation.Commit(
        view,                // owner
        "TranslateXTo",      // name
        16,                  // rate
        length,              // length
        null,                // easing
        (finalValue, cancelled) => taskCompletionSource.SetResult(cancelled)); // finished

   return taskCompletionSource.Task;
}
```

Notice that the current value of the `TranslationX` property is passed to the `Animation` constructor for the `start` argument, and the `x` parameter to `TranslateXTo` is passed as the `end` argument. The `TaskCompletionSource` has a type argument of `bool` so that the method can indicate if it's been cancelled or not. The method returns the `Task` property of the `TaskCompletionSource` object and calls `SetResult` in the `finished` callback of the `Commit` method.

However, there is a subtle flaw in this `TranslateXTo` method. What happens if the visual element being animated is removed from the visual tree during the course of the animation? In theory, if there are no other references to that object, it should become eligible for garbage collection. However, there *will* be a reference to that object in the animation method. The element will continue to be animated—and prevented from being garbage collected—even though there are no other references to that element!

You can avoid this peculiar situation if the animation method creates a `WeakReference` object to the animated element. The `WeakReference` allows the animation method to refer to the element but does not increase the reference count for purposes of garbage collection. While this is something you don't need to bother with for animation methods in your own application—because you're probably aware when elements are removed from visual trees—it's something you should probably do in any animation method that appears in a library.

The `TranslateXTo` method is in the **Xamarin.FormsBook.Toolkit** library, so it includes the use of `WeakReference`. Because the element could be gone when the callback method is called, the method must get a reference to the element with the `TryGetTarget` method. That method returns `false` if the object is no longer available:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class MoreViewExtensions
    {
        public static Task<bool> TranslateXTo(this VisualElement view, double x,
                                              uint length = 250, Easing easing = null)
        {
            easing = easing ?? Easing.Linear;
            TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();
            WeakReference<VisualElement> weakViewRef = new WeakReference<VisualElement>(view);

            Animation animation = new Animation(
                (value) =>
                    {
                        VisualElement viewRef;
                        if (weakViewRef.TryGetTarget(out viewRef))
                        {
                            viewRef.TranslationX = value;
                        }
                    },              // callback
                view.TranslationX,  // start
                x,                  // end
                easing);            // easing

            animation.Commit(
                view,               // owner
```

```
                "TranslateXTo",     // name
                16,                 // rate
                length,             // length
                null,               // easing
                (finalValue, cancelled) =>
                        taskCompletionSource.SetResult(cancelled)); // finished

           return taskCompletionSource.Task;
        }

        public static void CancelTranslateXTo(VisualElement view)
        {
            view.AbortAnimation("TranslateXTo");
        }
        …
}
```

Notice that a method to cancel the animation named "TranslateX" is also included.

This `TranslateXTo` method is demonstrated in the **SpringSlidingEntrance** program, which is the same as **SlidingEntrance** except that it has a reference to the **Xamarin.FormsBook.Toolkit** library and the `OnAppearing` override calls `TranslateXTo`:

```
public partial class SpringSlidingEntrancePage : ContentPage
{
    public SpringSlidingEntrancePage()
    {
        InitializeComponent();
    }

    async protected override void OnAppearing()
    {
        base.OnAppearing();

        double offset = 1000;

        foreach (View view in stackLayout.Children)
        {
            view.TranslationX = offset;
            offset *= -1;
        }

        foreach (View view in stackLayout.Children)
        {
            await Task.WhenAny(view.TranslateXTo(0, 1000, Easing.SpringOut),
                               Task.Delay(100));
        }
    }
}
```

The difference is, I'm sure you'll agree, well worth the effort. The elements on the page slide in and overshoot their destinations before settling into a well-ordered page.

The **Xamarin.FormsBook.Toolkit** library also has a `TranslateYTo` method that is basically the same as `TranslateXTo`, but with more concise syntax:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class MoreViewExtensions
    {
        …
        public static Task<bool> TranslateYTo(this VisualElement view, double y,
                                              uint length = 250, Easing easing = null)
        {
            easing = easing ?? Easing.Linear;
            TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();
            WeakReference<VisualElement> weakViewRef = new WeakReference<VisualElement>(view);

            Animation animation = new Animation((value) =>
                {
                    VisualElement viewRef;
                    if (weakViewRef.TryGetTarget(out viewRef))
                    {
                        viewRef.TranslationY = value;
                    }
                }, view.TranslationY, y, easing);

            animation.Commit(view, "TranslateYTo", 16, length, null,
                             (v, c) => taskCompletionSource.SetResult(c));

            return taskCompletionSource.Task;
        }

        public static void CancelTranslateYTo(VisualElement view)
        {
            view.AbortAnimation("TranslateYTo");
        }
        …
}
```

As a replacement for `TranslateTo`, you can use `TranslateXYTo`. As you learned earlier in this chapter, an `Easing` function that returns values less than 0 or greater than 1 shouldn't be passed to the `Commit` method for an animation with children. Instead, the `Easing` function should be passed to the `Animation` constructors of the children. This is what `TranslateXYTo` does:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class MoreViewExtensions
    {
        …
        public static Task<bool> TranslateXYTo(this VisualElement view, double x, double y,
                                               uint length = 250, Easing easing = null)
        {
            easing = easing ?? Easing.Linear;
            TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();
            WeakReference<VisualElement> weakViewRef = new WeakReference<VisualElement>(view);

            Action<double> callbackX = value =>
```

```
                {
                    VisualElement viewRef;
                    if (weakViewRef.TryGetTarget(out viewRef))
                    {
                        viewRef.TranslationX = value;
                    }
                };

            Action<double> callbackY = value =>
                {
                    VisualElement viewRef;
                    if (weakViewRef.TryGetTarget(out viewRef))
                    {
                        viewRef.TranslationY = value;
                    }
                };

            Animation animation = new Animation
            {
                { 0, 1, new Animation(callbackX, view.TranslationX, x, easing) },
                { 0, 1, new Animation(callbackY, view.TranslationY, y, easing) }
            };

            animation.Commit(view, "TranslateXYTo", 16, length, null,
                             (v, c) => taskCompletionSource.SetResult(c));

            return taskCompletionSource.Task;
        }

        public static void CancelTranslateXYTo(VisualElement view)
        {
            view.AbortAnimation("TranslateXYTo");
        }
        …
    }
}
```

Implementing a Bezier animation

Some graphics systems implement an animation that moves a visual object along a Bezier curve and even (optionally) rotates the visual object so it remains tangent to the curve.

The Bezier curve is named after Pierre Bézier, a French engineer and mathematician who developed the use of the curve in interactive computer-aided designs of automobile bodies while working at Renault. The curve is a type of spline defined by a start point and an end point and two control points. The curve passes through the start and end points but usually not the two control points. Instead, the control points function like “magnets” to pull the curve toward them.

In its two-dimensional form, the Bezier curve is represented mathematically as a pair of parametric cubic equations. Here is a `BezierSpline` structure in the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
```

```
    public struct BezierSpline
    {
        public BezierSpline(Point point0, Point point1, Point point2, Point point3)
            : this()
        {
            Point0 = point0;
            Point1 = point1;
            Point2 = point2;
            Point3 = point3;
        }

        public Point Point0 { private set; get; }

        public Point Point1 { private set; get; }

        public Point Point2 { private set; get; }

        public Point Point3 { private set; get; }

        public Point GetPointAtFractionLength(double t, out Point tangent)
        {
            // Calculate point on curve.
            double x = (1 - t) * (1 - t) * (1 - t) * Point0.X +
                       3 * t * (1 - t) * (1 - t) * Point1.X +
                       3 * t * t * (1 - t) * Point2.X +
                       t * t * t * Point3.X;

            double y = (1 - t) * (1 - t) * (1 - t) * Point0.Y +
                       3 * t * (1 - t) * (1 - t) * Point1.Y +
                       3 * t * t * (1 - t) * Point2.Y +
                       t * t * t * Point3.Y;

            Point point = new Point(x, y);

            // Calculate tangent to curve.
            x = 3 * (1 - t) * (1 - t) * (Point1.X - Point0.X) +
                6 * t * (1 - t) * (Point2.X - Point1.X) +
                3 * t * t * (Point3.X - Point2.X);

            y = 3 * (1 - t) * (1 - t) * (Point1.Y - Point0.Y) +
                6 * t * (1 - t) * (Point2.Y - Point1.Y) +
                3 * t * t * (Point3.Y - Point2.Y);

            tangent = new Point(x, y);
            return point;
        }
    }
}
```

The `Point0` and `Point3` points are the start and end points, while `Point1` and `Point2` are the two control points.

The `GetPointAtFractionLength` method returns the point on the curve corresponding to values of `t` ranging from 0 to 1. The first calculations of `x` and `y` in this method involve the standard parametric equations of the Bezier curve. When `t` is 0, the point on the curve is `Point0`, and when `t` is 1, the point on the curve is `Point3`.

`GetPointAtFractionLength` also has a second calculation of `x` and `y` based on the first derivative of the curve, so these values indicate the tangent of the curve at that point. Generally, we think of the tangent as a straight line that touches the curve but does not intersect it, so it might seem peculiar to express the tangent as another point. But this is not really a point. It's a vector in the direction from the point (0, 0) to the point (*x*, *y*). That vector can be turned into a rotation angle by using the inverse tangent function, also known as the arctangent, and available most conveniently to the .NET programmers as `Math.Atan2`, which has two arguments, `y` and `x` in that order, and returns an angle in radians. You'll need to convert to degrees for setting the `Rotation` property.

The `BezierPathTo` method in the **Xamarin.FormsBook.Toolkit** library moves the target visual element by calling the `Layout` method, which means that `BezierPathTo` is similar to `LayoutTo`. The method also optionally rotates the element by setting its `Rotation` property. Rather than splitting the job into two child animations, `BezierPathTo` does everything in the callback method of a single animation.

The start point of the Bezier curve is assumed to be the center of the visual element that the animation targets. The `BezierPathTo` method requires two control points and an end point. All points generated from the Bezier curve are also assumed to refer to the center of the visual element, so the points must be adjusted by half the element's width and height:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class MoreViewExtensions
    {
        …
        public static Task<bool> BezierPathTo(this VisualElement view,
                                              Point pt1, Point pt2, Point pt3,
                                              uint length = 250,
                                              BezierTangent bezierTangent = BezierTangent.None,
                                              Easing easing = null)
        {
            easing = easing ?? Easing.Linear;
            TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();
            WeakReference<VisualElement> weakViewRef = new WeakReference<VisualElement>(view);

            Rectangle bounds = view.Bounds;
            BezierSpline bezierSpline = new BezierSpline(bounds.Center, pt1, pt2, pt3);

            Action<double> callback = t =>
                {
                    VisualElement viewRef;
                    if (weakViewRef.TryGetTarget(out viewRef))
                    {
                        Point tangent;
                        Point point = bezierSpline.GetPointAtFractionLength(t, out tangent);
                        double x = point.X - bounds.Width / 2;
```

```
                        double y = point.Y - bounds.Height / 2;
                        viewRef.Layout(new Rectangle(new Point(x, y), bounds.Size));

                        if (bezierTangent != BezierTangent.None)
                        {
                            viewRef.Rotation = 180 * Math.Atan2(tangent.Y, tangent.X) / Math.PI;

                            if (bezierTangent == BezierTangent.Reversed)
                            {
                                viewRef.Rotation += 180;
                            }
                        }
                    }
                };

            Animation animation = new Animation(callback, 0, 1, easing);
            animation.Commit(view, "BezierPathTo", 16, length,
                finished: (value, cancelled) => taskCompletionSource.SetResult(cancelled));

            return taskCompletionSource.Task;
        }

        public static void CancelBezierPathTo(VisualElement view)
        {
            view.AbortAnimation("BezierPathTo");
        }
        …
    }
}
```

Applying the `Rotation` angle is still a bit tricky, however. If the points of a Bezier curve are defined so that the curve goes roughly from left to right across the screen, then the tangent is a vector that also goes from left to right, and the rotation of the animated element should preserve its orientation. But if the points of the Bezier curve go from right to left, then the tangent is also from right to left, and the mathematics dictate that the element should be flipped 180 degrees.

To control the orientation of the target element, a tiny enumeration is defined:

```
namespace Xamarin.FormsBook.Toolkit
{
    public enum BezierTangent
    {
        None,
        Normal,
        Reversed
    }
}
```

The `BezierPathTo` animation uses this to control how the tangent angle is applied to the `Rotation` property.

The **BezierLoop** program demonstrates the use of `BezierPathTo`. A `Button` sits in the upper-left corner of an `AbsoluteLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BezierLoop.BezierLoopPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <AbsoluteLayout>
        <Button Text="Click for Loop"
                Clicked="OnButtonClicked" />
    </AbsoluteLayout>
</ContentPage>
```

The `Clicked` handler for the `Button` begins by calculating the start and end points of the Bezier curve and the two control points. The start point is the upper-left corner where the `Button` initially sits. The end point is the upper-right corner. The two control points are the lower-right corner and the lower-left corner, respectively. This type of configuration actually creates a loop in the Bezier curve:

```
public partial class BezierLoopPage : ContentPage
{
    public BezierLoopPage()
    {
        InitializeComponent();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        Button button = (Button)sender;
        Layout parent = (Layout)button.Parent;

        // Center of Button in upper-left corner.
        Point point0 = new Point(button.Width / 2, button.Height / 2);

        // Lower-right corner of page.
        Point point1 = new Point(parent.Width, parent.Height);

        // Lower-left corner of page.
        Point point2 = new Point(0, parent.Height);

        // Center of Button in upper-right corner.
        Point point3 = new Point(parent.Width - button.Width / 2, button.Height / 2);

        // Initial angle of Bezier curve (vector from Point0 to Point1).
        double angle = 180 / Math.PI * Math.Atan2(point1.Y - point0.Y,
                                                  point1.X - point0.X);

        await button.RotateTo(angle, 1000, Easing.SinIn);

        await button.BezierPathTo(point1, point2, point3, 5000,
                                  BezierTangent.Normal, Easing.SinOut);

        await button.BezierPathTo(point2, point1, point0, 5000,
                                  BezierTangent.Reversed, Easing.SinIn);
```

```
        await button.RotateTo(0, 1000, Easing.SinOut);
    }
}
```

The tangent to the Bezier curve at its very beginning is the line from `point0` to `point1`. This is the `angle` variable that the method calculates so it can first use `RotateTo` to rotate the `Button` to avoid a jump when the `BezierPathTo` animation begins. The first `BezierPathTo` moves the `Button` from the upper-left corner to the upper-right corner with a loop near the bottom of the screen:

A second `BezierPathTo` then reverses the trip back to the upper-left corner. (This is where the `BezierTangent` enumeration comes into play. Without it, the `Button` suddenly flips upside down as the second `BezierPathTo` begins.) A final `RotateTo` restores it to its original orientation.

Working with AnimationExtensions

Why does `ViewExtensions` not include a `ColorTo` animation? There are three plausible reasons why such a method isn't as obvious as you might initially assume:

Firstly, the only `Color` property defined by `VisualElement` is `BackgroundColor`, but that's usually not the `Color` property you want to animate. It's more likely you want to animate the `TextColor` property of `Label` or the `Color` property of `BoxView`.

Secondly, all the methods in `ViewExtensions` animate a property from its current value to a specified value. But often the current value of a property of type `Color` is `Color.Default`, which is not a real color and which cannot be used in an interpolation calculation.

Thirdly, the interpolation between two `Color` values can be calculated in a variety of different ways, but two stand out as the most likely: You might want to interpolate the red-green-blue values or the hue-saturation-luminosity values. The intermediate values will be different in these two cases.

Let's take care of these three problems with three different solutions:

Firstly, let's not have the color-animation method target a particular property. Let's write the method with a callback method that passes the interpolated `Color` value back to the caller.

Secondly, let's require that both a start `Color` value and an end `Color` value be supplied to the animation method.

Thirdly, let's write two different methods, `RgbColorAnimation` and `HslColorAnimation`.

You could certainly use the `Animation` class and `Commit` for this job, but let's instead dive deeper into the Xamarin.Forms animation system and use a method in the `AnimationExtensions` class.

`AnimationExtensions` has four different methods named `Animate`, as well as an `AnimateKinetic` method. The `AnimateKinetic` method is intended to apply a "drag" value to an animation so that it slows down as if by friction. However, it's not yet working in a way that allows the results to be easily predicted, and it is not demonstrated in this chapter.

Of the four `Animate` methods, the generic form is the most versatile:

```
public static void Animate<T>(this IAnimatable self,
                              string name,
                              Func<double, T> transform,
                              Action<T> callback,
                              uint rate = 16,
                              uint length = 250,
                              Easing easing = null,
                              Action<T, bool> finished = null,
                              Func<bool> repeat = null);
```

The generic type is the type of the property you want to animate—for example, `Color`. By this time you should recognize all these parameters except for the callback method named `transform`. The input to that callback is always a `t` or `progress` value ranging from 0 to 1. The output is a value of the generic type—for example, `Color`. That value is then passed to the `callback` method for application to a particular property.

Here are `RgbColorAnimation` and `HslColorAnimation` in the `MoreViewExtensions` class of the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public static class MoreViewExtensions
    {
        …
        public static Task<bool> RgbColorAnimation(this VisualElement view,
                                                   Color fromColor, Color toColor,
                                                   Action<Color> callback,
                                                   uint length = 250,
```

```
                                                Easing easing = null)
        {
            Func<double, Color> transform = (t) =>
                {
                    return Color.FromRgba(fromColor.R + t * (toColor.R - fromColor.R),
                                          fromColor.G + t * (toColor.G - fromColor.G),
                                          fromColor.B + t * (toColor.B - fromColor.B),
                                          fromColor.A + t * (toColor.A - fromColor.A));
                };

            return ColorAnimation(view, "RgbColorAnimation", transform,
                                  callback, length, easing);
        }

        public static void CancelRgbColorAnimation(VisualElement view)
        {
            view.AbortAnimation("RgbColorAnimation");
        }

        public static Task<bool> HslColorAnimation(this VisualElement view,
                                                   Color fromColor, Color toColor,
                                                   Action<Color> callback,
                                                   uint length = 250,
                                                   Easing easing = null)
        {
            Func<double, Color> transform = (t) =>
            {
                return Color.FromHsla(
                    fromColor.Hue + t * (toColor.Hue - fromColor.Hue),
                    fromColor.Saturation + t * (toColor.Saturation - fromColor.Saturation),
                    fromColor.Luminosity + t * (toColor.Luminosity - fromColor.Luminosity),
                    fromColor.A + t * (toColor.A - fromColor.A));
            };

            return ColorAnimation(view, "HslColorAnimation", transform,
                                  callback, length, easing);
        }

        public static void CancelHslColorAnimation(VisualElement view)
        {
            view.AbortAnimation("HslColorAnimation");
        }

        static Task<bool> ColorAnimation(VisualElement view,
                                         string name,
                                         Func<double, Color> transform,
                                         Action<Color> callback,
                                         uint length,
                                         Easing easing)
        {
            easing = easing ?? Easing.Linear;
            TaskCompletionSource<bool> taskCompletionSource = new TaskCompletionSource<bool>();

            view.Animate<Color>(name, transform, callback, 16,
```

```
                                length, easing, (value, canceled) =>
                    {
                        taskCompletionSource.SetResult(canceled);
                    });

                return taskCompletionSource.Task;
            }
        }
}
```

The two methods define their own `transform` functions and then make use of the private `ColorAnimation` method to actually make the call to the `Animate` method in `AnimationExtensions`. Because these methods don't explicitly target a particular visual element, there is no need for the `WeakReference` class.

The **ColorAnimations** program demonstrates these methods for animating various color properties in various ways. The XAML file as a `Label`, two `Button` elements, and two `BoxView` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ColorAnimations.ColorAnimationsPage">
    <StackLayout>
        <Label x:Name="label"
               Text="TEXT"
               FontSize="48"
               FontAttributes="Bold"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Rainbow Background"
                Clicked="OnRainbowBackgroundButtonClicked"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand" />

        <Button Text="BoxView Color"
                Clicked="OnBoxViewColorButtonClicked"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand" />

        <StackLayout Orientation="Horizontal">
            <BoxView x:Name="boxView1"
                     Color="Blue"
                     HeightRequest="100"
                     HorizontalOptions="FillAndExpand" />

            <BoxView x:Name="boxView2"
                     Color="Blue"
                     HeightRequest="100"
                     HorizontalOptions="FillAndExpand" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The code-behind file uses a mix of `RgbColorAnimation` and `HslColorAnimation` to animate the colors of the `Label` text and its background, the background of the page, and the two `BoxView` elements.

The `Label` text and its background are continuously animated oppositely between black and white. Only midway through the animations—when both the text and the background are medium gray—is the text invisible:

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

        AnimationLoop();
    }

    async void AnimationLoop()
    {
        while (true)
        {
            Action<Color> textCallback = color => label.TextColor = color;
            Action<Color> backCallback = color => label.BackgroundColor = color;

            await Task.WhenAll(
                    label.RgbColorAnimation(Color.White, Color.Black, textCallback, 1000),
                    label.HslColorAnimation(Color.Black, Color.White, backCallback, 1000));

            await Task.WhenAll(
                    label.RgbColorAnimation(Color.Black, Color.White, textCallback, 1000),
                    label.HslColorAnimation(Color.White, Color.Black, backCallback, 1000));
        }
    }
    …
}
```

When animating between `Color.Black` and `Color.White`, it doesn't matter whether you use `RgbColorAnimation` or `HslColorAnimation`. The result is the same. Black is represented in RGB as (0, 0, 0) and in HSL as (0, 0, 0). White is (1, 1, 1) in RGB and (0, 0, 1) in HSL. At the midway point, the RGB color (0.5, 0.5, 0.5) is the same as the HSL color (0, 0, 0.5).

The `HslColorAnimation` is great for animating through all the hues, which roughly correspond to the colors of the rainbow, traditionally red, orange, yellow, green, blue, indigo, and violet. In color animations, a final animation back to red usually occurs at the end. Animating RGB colors through this sequence requires first animating from `Color.Red` to `Color.Yellow`, then `Color.Yellow` to `Color.Green`, then `Color.Green` to `Color.Aqua`, then `Color.Aqua` to `Color.Blue`, then `Color.Blue` to `Color.Fuchsia`, and finally `Color.Fuchsia` to `Color.Red`.

With `HslColorAnimation`, all that's necessary is to animate between two representations of red, one with the `Hue` set to 0 and the other with the `Hue` set to 1:

```
public partial class ColorAnimationsPage : ContentPage
{
    …
    async void OnRainbowBackgroundButtonClicked(object sender, EventArgs args)
    {
        // Animate from Red to Red.
        await this.HslColorAnimation(Color.FromHsla(0, 1, 0.5),
                                     Color.FromHsla(1, 1, 0.5),
                                     color => BackgroundColor = color,
                                     10000);

        BackgroundColor = Color.Default;
    }
    …
}
```

Even with simple animations between two primary colors, `RgbColorAnimation` and `HslColorAnimation` can produce different results. Consider an animation from blue to red. The **ColorAnimations** program demonstrates the difference by animating the colors of two `BoxView` elements with the two animation methods:

```
public partial class ColorAnimationsPage : ContentPage
{
    …
    async void OnBoxViewColorButtonClicked(object sender, EventArgs args)
    {
        Action<Color> callback1 = color => boxView1.Color = color;
        Action<Color> callback2 = color => boxView2.Color = color;

        await Task.WhenAll(boxView1.RgbColorAnimation(Color.Blue, Color.Red, callback1, 2000),
                           boxView2.HslColorAnimation(Color.Blue, Color.Red, callback2, 2000));

        await Task.WhenAll(boxView1.RgbColorAnimation(Color.Red, Color.Blue, callback1, 2000),
                           boxView2.HslColorAnimation(Color.Red, Color.Blue, callback2, 2000));

    }
}
```

Blue has an RGB representation of (0, 0, 1) and an HSL representation of (0.67, 1, 0.5). Red has an RGB representation of (1, 0, 0) and in HSL is (1, 0, 0.5). Halfway through the RGB animation, the interpolated color is (0.5, 0, 0.5), which is known in Xamarin.Forms as `Color.Magenta`. However, midway through the `HslColorAnimation`, the interpolated color is (0.83, 1, 0.5), which is the lighter `Color.Fuchsia`, which has an RGB representation of (1, 0, 1).

This screenshot shows the progress (from left to right) of the animation of the two `BoxView` elements from blue to red:

Neither is “right” or “wrong.” It’s just two different ways of interpolating between two colors, and the reason why a simple `ColorAnimation` method is inadequate.

Structuring your animations

There is no XAML representation of animations, so much of the focus of this chapter has necessarily been on code rather than markup.

However, when you’re using animations in conjunction with styles, and with MVVM and data binding, you’ll probably want a way to refer to animations in XAML. This is possible, and you’ll see in the next chapter how you can encapsulate animations within classes called *trigger actions* and *behaviors*, and then make them part of the styling and data binding of your application’s visuals.