half-grown size until it reaches the original width that's set in the XAML markup. On the other hand, if you run this code while no other animation is underway, nothing will happen. That's because the From value (the animated width) and the To value (the original width) are the same.

By

Instead of using To, you can use the By property. The By property is used to create an animation that changes a value *by* a set amount, rather than *to* a specific target. For example, you could create an animation that enlarges a button by 10 units more than its current size, as shown here:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.By = 10;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
cmdGrowIncrementally.BeginAnimation(Button.WidthProperty, widthAnimation);
```

This approach isn't necessary in the button example, because you could achieve the same result using a simple calculation to set the To property, like this:

```
widthAnimation.To = cmdGrowIncrementally.Width + 10;
```

However, the By value makes more sense when you're defining your animation in XAML, because XAML doesn't provide a way to perform simple calculations.

■ **Note** You can use By and From in combination, but it doesn't save you any work. The By value is simply added to the From value to arrive at the To value.

The By property is offered by most, but not all, animation classes that use interpolation. For example, it doesn't make sense with non-numeric data types, such as a Color structure (as used by ColorAnimation).

There's one other way to get similar behavior without using the By property—you can create an *additive* animation by setting the IsAdditive property. When you do, the current value is added to both the From and To values automatically. For example, consider this animation:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 0;
widthAnimation.To = -10;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
widthAnimation.IsAdditive = true;
```

It starts from the current value and finishes at a value that's reduced by 10 units. On the other hand, if you use this animation:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 10;
widthAnimation.To = 50;
widthAnimation.Duration = TimeSpan.FromSeconds(0.5);
widthAnimation.IsAdditive = true;
```

the property jumps to the new value (which is 10 units greater than the current value) and then increases until it reaches a final value that is 50 more units than the current value before the animation began.

Duration

The Duration property is straightforward enough—it takes the time interval (in milliseconds, minutes, hours, or whatever else you'd like to use) between the time the animation starts and the time it ends. Although the duration of the animations in the previous examples is set using a TimeSpan, the Duration property actually requires a Duration object. Fortunately, Duration and TimeSpan are quite similar, and the Duration structure defines an implicit cast that can convert System.TimeSpan to System.Windows.Duration as needed. That's why this line of code is perfectly reasonable:

```
widthAnimation.Duration = TimeSpan.FromSeconds(5);
```

So, why bother introducing a whole new type? The Duration also includes two special values that can't be represented by a TimeSpan object—Duration.Automatic and Duration.Forever. Neither of these values is useful in the current example. (Automatic simply sets the animation to a one-second duration, and Forever makes the animation infinite in length, which prevents it from having any effect.) However, these values become useful when creating more complex animations.

Simultaneous Animations

You can use BeginAnimation() to launch more than one animation at a time. The BeginAnimation() method returns almost immediately, allowing you to use code like this to animate two properties simultaneously:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.From = 160;
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);

DoubleAnimation heightAnimation = new DoubleAnimation();
heightAnimation.From = 40;
heightAnimation.To = this.Height - 50;
heightAnimation.Duration = TimeSpan.FromSeconds(5);

cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
cmdGrow.BeginAnimation(Button.HeightProperty, heightAnimation);
```

In this example, the two animations are not synchronized. That means the width and height won't grow at exactly the same intervals. (Typically, you'll see the button grow wider and then grow taller just after.) You can overcome this limitation by creating animations that are bound to the same timeline. You'll learn this technique later in this chapter, when you consider storyboards.

Animation Lifetime

Technically, WPF animations are *temporary*, which means they don't actually change the value of the underlying property. While an animation is active, it simply overrides the property value. This is because

of the way that dependency properties work (as described in Chapter 4), and it's an often overlooked detail that can cause significant confusion.

A one-way animation (like the button growing animation) remains active after it finishes running. That's because the animation needs to hold the button's width at the new size. This can lead to an unusual problem—namely, if you try to modify the value of the property using code after the animation has completed, your code will appear to have no effect. That's because your code simply assigns a new local value to the property, but the animated value still takes precedence.

You can solve this problem in several ways, depending on what you're trying to accomplish:

- Create an animation that resets your element to its original state. You do this by not setting the To property. For example, the button shrinking animation reduces the width of the button to its last set size, after which you can change it in your code.
- Create a reversible animation. You do this by setting the AutoReverse property to true. For example, when the button growing animation finishes widening the button, it will play out the animation in reverse, returning it to its original width. The total duration of your animation will be doubled.
- Change the FillBehavior property. Ordinarily, FillBehavior is set to HoldEnd, which means that when an animation ends, it continues to apply its final value to the target property. If you change FillBehavior to Stop, as soon as the animation ends the property reverts to its original value.
- Remove the animation object when the animation is complete by handling the Completed event of the animation object.

The first three options change the behavior of your animation. One way or another, they return the animated property to its original value. If this isn't what you want, you need to use the last option.

First, before you launch the animation, attach an event handler that reacts when the animation finishes:

```
widthAnimation.Completed += animation_Completed;
```

■ **Note** The Completed event is a normal .NET event that takes an ordinary EventArgs object with no additional information. It's not a routed event.

When the Completed event fires, you can render the animation inactive by calling the BeginAnimation() method. You simply need to specify the property and pass in a null reference for the animation object:

```
cmdGrow.BeginAnimation(Button.WidthProperty, null);
```

When you call BeginAnimation(), the property returns to the value it had before the animation started. If this isn't what you want, you can take note of the current value that's being applied by the animation, remove the animation, and then manually set the new property, like so:

```
double currentWidth = cmdGrow.Width;
cmdGrow.BeginAnimation(Button.WidthProperty, null);
cmdGrow.Width = currentWidth;
```

Keep in mind that this changes the local value of the property. That may affect how other animations work. For example, if you animate this button with an animation that doesn't specify the From property, it uses this newly applied value as a starting point. In most cases, this is the behavior you want.

The Timeline Class

As you've seen, every animation revolves around a few key properties. You've seen several of these properties: From and To (which are provided in animation classes that use interpolation) and Duration and FillBehavior (which are provided in all animation classes). Before going any further, it's worth taking a closer look at the properties you have to work with.

Figure 15-2 shows the inheritance hierarchy of the WPF animation types. It includes all the base classes, but it leaves out the full 42 animation types (and the corresponding *TypeName*AnimationBase classes).

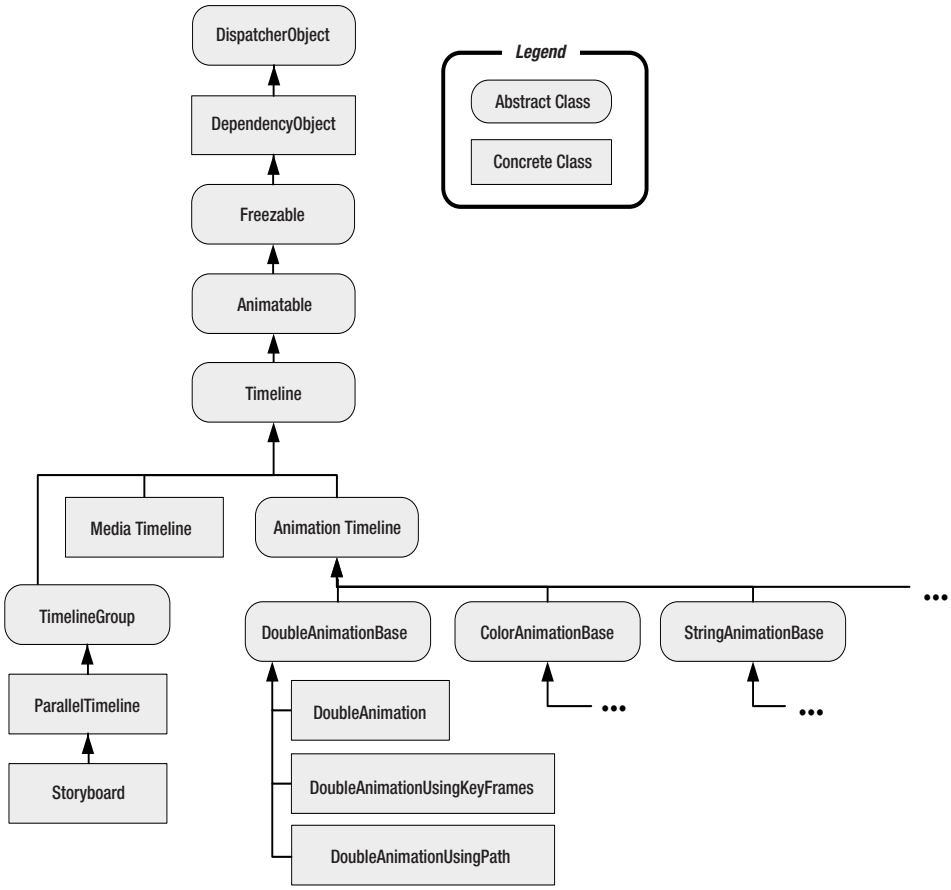

Figure 15-2. *The animation class hierarchy*

The class hierarchy includes three main branches that derive from the abstract Timeline class. MediaTimeline is used when playing audio or video files—it's described in Chapter 26. AnimationTimeline is used for the property-based animation system you've considered so far. And TimelineGroup allows you to synchronize timelines and control their playback. It's described later in this chapter in the "Synchronized Animations" section, when you tackle storyboards.

The first useful members appear in the Timeline class, which defines the Duration property you've already considered and a few more. Table 15-1 lists its properties.

Table 15-1. *Timeline Properties*

Name	Description
BeginTime	Sets a delay that will be added before the animation starts (as a TimeSpan). This delay is added to the total time, so a five-second animation with a five-second delay takes ten seconds. BeginTime is useful when synchronizing different animations that start at the same time but should apply their effects in sequence.
Duration	Sets the length of time the animation runs, from start to finish, as a Duration object.
SpeedRatio	Increases or decreases the speed of the animation. Ordinarily, SpeedRatio is 1. If you increase it, the animation completes more quickly (for example, a SpeedRatio of 5 completes five times faster). If you decrease it, the animation is slowed down (for example, a SpeedRatio of 0.5 takes twice as long). You can change the Duration of your animation for an equivalent result. The SpeedRatio is not taken into account when applying the BeginTime delay.
AccelerationRatio and DecelerationRatio	Makes an animation nonlinear, so it starts off slow and then speeds up (by increasing the AccelerationRatio) or slows down at the end (by increasing the DecelerationRatio). Both values are set from 0 to 1 and begin at 0. Furthermore, the total of both values cannot exceed 1.
AutoReverse	If true, the animation will play out in reverse once it's complete, reverting to the original value. This also doubles the time the animation takes. If you've increased the SpeedRatio, it applies to both the initial playback of the animation and the reversal. The BeginTime applies only to the very beginning of the animation—it doesn't delay the reversal.
FillBehavior	Determines what happens when the animation ends. Usually, it keeps the property fixed at the ending value (FillBehavior.HoldEnd), but you can also choose to return it to its original value (FillBehavior.Stop).
RepeatBehavior	Allows you to repeat an animation a specific number of times or for a specific time interval. The RepeatBehavior object that you use to set this property determines the exact behavior.

Although BeginTime, Duration, SpeedRatio, and AutoReverse are all fairly straightforward, some of the other properties warrant closer examination. The following sections delve into AccelerationRatio, DecelerationRatio, and RepeatBehavior.

AccelerationRatio and DecelerationRatio

AccelerationRatio and DecelerationRatio allow you to compress part of the timeline so it passes by more quickly. The rest of the timeline is stretched to compensate so that the total time is unchanged.

Both of these properties represent a percentage value. For example, an AccelerationRatio of 0.3 indicates that you want to spend the first 30% of the duration of the animation accelerating. For example, in a ten-second animation, the first three seconds would be taken up with acceleration, and the remaining seven seconds would pass at a consistent speed. (Obviously, the speed in the last seven seconds is faster than the speed of a nonaccelerated animation, because it needs to make up for the slow start.) If you set AccelerationRatio to 0.3 and DecelerationRatio to 0.3, acceleration takes place for the first three seconds, the middle four seconds are at a fixed maximum speed, and deceleration takes place for the last three seconds. Viewed this way, it's obvious that the total of AccelerationRatio and DecelerationRatio can't top 1, because then it required more than 100% of the available time to perform the requested acceleration and deceleration. Of course, you could set AccelerationRatio to 1 (in which case the animation speeds up from start to finish) or DecelerationRatio to 1 (in which case the animation slows down from start to finish).

Animations that accelerate and decelerate are often used to give a more natural appearance. However, the AccelerationRatio and DecelerationRatio give you only relatively crude control. For example, they don't let you vary the acceleration or set it specifically. If you want to have an animation that uses varying degrees of acceleration, you'll need to define a series of animations, one after the other, and set the AccelerationRatio and DecelerationRatio property of each one, or you'll need to use a key frame animation with key spline frames (as described in Chapter 16). Although this technique gives you plenty of flexibility, keeping track of all the details is a headache, and it's a perfect case for using a design tool to construct your animations.

RepeatBehavior

The RepeatBehavior property allows you to control how an animation is repeated. If you want to repeat it a fixed number of times, pass the appropriate number of times to the RepeatBehavior constructor. For example, this animation repeats twice:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
widthAnimation.RepeatBehavior = new RepeatBehavior(2);
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

When you run this animation, the button will increase in size (over five seconds), jump back to its original value, and then increase in size again (over five seconds), ending at the full width of the window. If you've set AutoReverse to true, the behavior is slightly different—the entire animation is completed forward and backward (meaning the button expands and then shrinks), and *then* it's repeated again.

■ **Note** Animations that use interpolation provide an IsCumulative property, which tells WPF how to deal with each repetition. If IsCumulative is true, the animation isn't repeated from start to finish. Instead, each subsequent animation adds to the previous one. For example, if you use IsCumulative with the animation shown earlier, the button will expand twice as wide over twice as much time. To put it another way, the first iteration is treated normally, but every repetition after that is treated as though you set IsAdditive to true.

Rather than using RepeatBehavior to set a repeat count, you can use it to set a repeat *interval*. To do so, simply pass a TimeSpan to the RepeatBehavior constructor. For example, the following animation repeats itself for 13 seconds:

```
DoubleAnimation widthAnimation = new DoubleAnimation();
widthAnimation.To = this.Width - 30;
widthAnimation.Duration = TimeSpan.FromSeconds(5);
widthAnimation.RepeatBehavior = new RepeatBehavior(TimeSpan.FromSeconds(13));
cmdGrow.BeginAnimation(Button.WidthProperty, widthAnimation);
```

In this example, the Duration property specifies that the entire animation takes five seconds. As a result, the RepeatBehavior of 13 seconds will trigger two repeats and then leave the button halfway through a third repeat (at the three-second mark).

■ **Tip** You can use RepeatBehavior to perform just part of an animation. To do so, use a fractional number of repetitions, or use a TimeSpan that's less than the duration.

Finally, you can cause an animation to repeat itself endlessly with the RepeatBehavior.Forever value:

```
widthAnimation.RepeatBehavior = RepeatBehavior.Forever;
```

Storyboards

As you've seen, WPF animations are represented by a group of animation classes. You set the relevant information, such as the starting value, ending value, and duration, using a handful of properties. This obviously makes them a great fit for XAML. What's less clear is how you wire an animation up to a particular element and property and how you trigger it at the right time.

It turns out that two ingredients are at work in any declarative animation:

- **A storyboard.** It's the XAML equivalent of the BeginAnimation() method. It allows you to direct an animation to the right element and property.
- **An event trigger.** It responds to a property change or event (such as the Click event of a button) and controls the storyboard. For example, to start an animation, the event trigger must *begin* the storyboard.

You'll learn how both pieces work in the following sections.

The Storyboard

A storyboard is an enhanced timeline. You can use it to group multiple animations, and it also has the ability to control the playback of animation—pausing it, stopping it, and changing its position. However, the most basic feature provided by the Storyboard class is its ability to point to a specific property and specific element using the TargetProperty and TargetName properties. In other words, the storyboard bridges the gap between your animation and the property you want to animate.

Here's how you might define a storyboard that manages a DoubleAnimation:

```
<Storyboard TargetName="cmdGrow" TargetProperty="Width">
  <DoubleAnimation From="160" To="300" Duration="0:0:5"></DoubleAnimation>
</Storyboard>
```

Both TargetName and TargetProperty are attached properties. That means you can apply them directly to the animation, as shown here:

```
<Storyboard>
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   From="160" To="300" Duration="0:0:5"></DoubleAnimation>
</Storyboard>
```

This syntax is more common, because it allows you to put several animations in the same storyboard but allow each animation to act on a different element and property.

Defining a storyboard is the first step to creating an animation. To actually put this storyboard into action, you need an event trigger.

Event Triggers

You first learned about event triggers in Chapter 11, when you considered styles. Styles give you one way to attach an event trigger to an element. However, you can define an event trigger in four places:

- In a style (the Styles.Triggers collection)
- In a data template (the DataTemplate.Triggers collection)
- In a control template (the ControlTemplate.Triggers collection)
- In an element directly (the FrameworkElement.Triggers collection)

When creating an event trigger, you need to indicate the routed event that starts the trigger and the action (or actions) that are performed by the trigger. With animations, the most common action is BeginStoryboard, which is equivalent to calling BeginAnimation().

The following example uses the Triggers collection of a button to attach an animation to the Click event. When the button is clicked, it grows.

```
<Button Padding="10" Name="cmdGrow" Height="40" Width="160"
 HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button.Triggers>
```

```
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="Width"
             To="300" Duration="0:0:5"></DoubleAnimation>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>

  <Button.Content>
    Click and Make Me Grow
  </Button.Content>
</Button>
```

■ **Tip** To create an animation that fires when the window first loads, add an event trigger in the Window.Triggers collection that responds to the Window.Loaded event.

The Storyboard.TargetProperty property identifies the property you want to change (in this case, Width). If you don't supply a class name, the storyboard uses the parent element, which is the button you want to expand. If you want to set an attached property (for example, Canvas.Left or Canvas.Top), you need to wrap the entire property in brackets, like this:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Left)" ... />
```

The Storyboard.TargetName property isn't required in this example. When you leave it out, the storyboard uses the parent element, which is the button.

■ **Note** All an event trigger is able to do is launch *actions*. All actions are represented by classes that derive from System.Windows.TriggerAction. Currently, WPF includes a very small set of actions that are designed for interacting with a storyboard and controlling media playback.

There's one difference between the declarative approach shown here and the code-only approach demonstrated earlier. Namely, the To value is hard-coded at 300 units, rather than set relative to the size of the containing window. If you wanted to use the window width, you'd need to use a data binding expression, like so:

```
<DoubleAnimation Storyboard.TargetProperty="Width"
 To="{Binding ElementName=window,Path=Width}" Duration="0:0:5">
</DoubleAnimation>
```

This still doesn't get exactly the result you need. Here, the button grows from its current size to the full width of the window. The code-only approach enlarges the button to 30 units less than the full size, using a trivial calculation. Unfortunately, XAML doesn't support inline calculations. One solution is to build an IValueConverter that does the work for you. Fortunately, this odd trick is easy to implement (and many developers have). You can find one example at `http://tinyurl.com/y9lglyu`, or check out the downloadable examples for this chapter.

■ **Note** Another option is to create a custom dependency property in your window class that performs the calculation. You can then bind your animation to the custom dependency property. For more information about creating dependency properties, see Chapter 4.

You can now duplicate all the examples you've seen so far by creating triggers and storyboards and setting the appropriate properties of the DoubleAnimation object.

Attaching Triggers with a Style

The FrameworkElement.Triggers collection is a bit of an oddity. It supports only event triggers. The other trigger collections (Styles.Triggers, DataTemplate.Triggers, and ControlTemplate.Triggers) are more capable. They support the three basic types of WPF triggers: property triggers, data triggers, and event triggers.

■ **Note** There's no technical reason why the FrameworkElement.Triggers collection shouldn't support additional trigger types, but this functionality wasn't implemented in time for the first version of WPF.

Using an event trigger is the most common way to attach an animation. However, it's not your only option. If you're using the Triggers collection in a style, data template, or control template, you can also create a property trigger that reacts when a property value changes. For example, here's a style that duplicates the example shown earlier. It triggers a storyboard when IsPressed is true:

```
<Window.Resources>
  <Style x:Key="GrowButtonStyle">
    <Style.Triggers>
      <Trigger Property="Button.IsPressed" Value="True">
        <Trigger.EnterActions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation Storyboard.TargetProperty="Width"
               To="250" Duration="0:0:5"></DoubleAnimation>
            </Storyboard>
          </BeginStoryboard>
        </Trigger.EnterActions>
      </Trigger>
```

```
    </Style.Triggers>
  </Style>
</Window.Resources>
```

You can attach actions to a property trigger in two ways. You can use Trigger.EnterActions to set actions that will be performed when the property changes to the value you specify (in the previous example, when IsPressed becomes true) and use Trigger.ExitActions to set actions that will be performed when the property changes back (when the value of IsPressed returns False false). This is a handy way to wrap together a pair of complementary animations.

Here's the button that uses the style shown earlier:

```
<Button Padding="10" Name="cmdGrow" Height="40" Width="160"
 Style="{StaticResource GrowButtonStyle}"
 HorizontalAlignment="Center" VerticalAlignment="Center">
 Click and Make Me Grow
</Button>
```

Remember, you don't need to use property triggers in a style. You can also use event triggers, as you saw in the previous section. Finally, you don't need to define a style separately from the button that uses it (you can set the Button.Style property with an inline style), but this two-part separation is more common, and it gives you the flexibility to apply the same animation to multiple elements.

■ **Note** Triggers are also handy when you fuse them into a control template, which allows you to add visual pizzazz to a standard WPF control. Chapter 17 shows numerous examples of control templates that use animations, including a ListBox that animates its child items with triggers.

Overlapping Animations

The storyboard gives you the ability to change the way you deal with animations that overlap—in other words, when a second animation is applied to a property that is already being animated. You do this using the BeginStoryboard.HandoffBehavior property.

Ordinarily, when two animations overlap, the second animation overrides the first one immediately. This behavior is known as *snapshot-and-replace* (and represented by the SnapshotAndReplace value in the HandoffBehavior enumeration). When the second animation starts, it takes a snapshot of the property as it currently is (based on the first animation), stops the animation, and replaces it with the new animation.

The only other HandoffBehavior option is Compose, which fused the second animation into the first animation's timeline. For example, consider a revised version of the ListBox example that uses HandoffBehavior.Compose when shrinking the button:

```
<EventTrigger RoutedEvent="ListBoxItem.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard HandoffBehavior="Compose">
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="FontSize"
         BeginTime="0:0:0.5" Duration="0:0:0.2"></DoubleAnimation>
      </Storyboard>
```

```
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

Now, if you move the mouse onto a ListBoxItem and off it, you'll see a different behavior. When you move the mouse off the item, it will continue expanding, which will be clearly visible until the second animation reaches its begin time delay of 0.5 seconds. Then, the second animation will shrink the button. Without the Compose behavior, the button would simply wait, fixed at its current size, for the 0.5-second time interval before the second animation kicks in.

Using a HandoffBehavior of compose requires more overhead. That's because the clock that's used to run the original animation won't be released when the second animation starts. Instead, it will stay alive until the ListBoxItem is garbage collected or a new animation is used on the same property.

■ **Tip** If performance becomes an issue, the WPF team recommends that you manually release the animation clock for your animations as soon as they are complete (rather than waiting for the garbage collector to find them). To do this, you need to handle an event like Storyboard.Completed. Then, call BeginAnimation() on the element that has just finished its animation, supplying the appropriate property and a null reference in place of an animation.

Synchronized Animations

The Storyboard class derives indirectly from TimelineGroup, which gives it the ability to hold more than one animation. Best of all, these animations are managed as one group—meaning they're started at the same time.

To see an example, consider the following storyboard. It starts two animations, one that acts on the Width property of a button and the other that acts on the Height property. Because the animations are grouped into one storyboard, they increment the button's dimensions in unison, which gives a more synchronized effect than simply calling BeginAnimation() multiple times in your code.

```
<EventTrigger RoutedEvent="Button.Click">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="Width"
         To="300" Duration="0:0:5"></DoubleAnimation>
        <DoubleAnimation Storyboard.TargetProperty="Height"
         To="300" Duration="0:0:5"></DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
```

In this example, both animations have the same duration, but this isn't a requirement. The only consideration with animations that end at different times is their FillBehavior. If an animation's FillBehavior property is set to HoldEnd, it holds the value until all the animations in the storyboard are completed. If the storyboard's FillBehavior property is HoldEnd, the final animated values are held indefinitely (until a new animation replaces this one or until you manually remove the animation).

It's at this point that the Timeline properties you learned about in Table 15-1 start to become particularly useful. For example, you can use SpeedRatio to make one animation in a storyboard run faster than the other. Or, you can use BeginTime to offset one animation relative to another so that it starts at a specific point.

■ **Note** Because Storyboard derives from Timeline, you can use all the properties that were described in Table 15-1 to configure its speed, use acceleration or deceleration, introduce a delay time, and so on. These properties will affect all the contained animations, and they're cumulative. For example, if you set the Storyboard.SpeedRatio to 2 and the DoubleAnimation.SpeedRatio to 2, that animation will run four times faster than usual.

Controlling Playback

So far, you've been using one action in your event triggers—the BeginStoryboard action that launches an animation. However, you can use several other actions to control a storyboard once it's created. These actions, which derive from the ControllableStoryboardAction class, are listed in Table 15-2.

Table 15-2. *Action Classes for Controlling a Storyboard*

Name	Description
PauseStoryboard	Stops playback of an animation and keeps it at the current position.
ResumeStoryboard	Resumes playback of a paused animation.
StopStoryboard	Stops playback of an animation and resets the animation clock to the beginning.
SeekStoryboard	Jumps to a specific position in an animation's timeline. If animation is currently playing, it continues playback from the new position. If the animation is currently paused, it remains paused.
SetStoryboardSpeedRatio	Changes the SpeedRatio of the entire storyboard (rather than just one animation inside).
SkipStoryboardToFill	Moves the storyboard to the end of its timeline. Technically, this period is known as the *fill region*. For a standard animation, with FillBehavior set to HoldEnd, the animation continues to hold the final value.
RemoveStoryboard	Removes a storyboard, halting any in-progress animation and returning the property to its original, last-set value. This has the same effect as calling BeginAnimation() on the appropriate element with a null animation object.

■ **Note** Stopping an animation is not equivalent to completing an animation (unless FillBehavior is set to Stop). That's because even when an animation reaches the end of its timeline, it continues to apply its final value. Similarly, when an animation is paused, it continues to apply the most recent intermediary value. However, when an animation is stopped, it no longer applies any value, and the property reverts to its preanimation value.

There's an undocumented stumbling block to using these actions. For them to work successfully, you must define all the triggers in one Triggers collection. If you place the BeginStoryboard action in a different trigger collection than the PauseStoryboard action, the PauseStoryboard action won't work. To see the design you need to use, it helps to consider an example.

For example, consider the window shown in Figure 15-3. It superimposes two Image elements in exactly the same position, using a grid. Initially, only the topmost image—which shows a day scene of a Toronto city landmark—is visible. But as the animation runs, it reduces the opacity from 1 to 0, eventually allowing the night scene to show through completely. The effect is as if the image is changing from day to night, like a sequence of time-lapse photography.

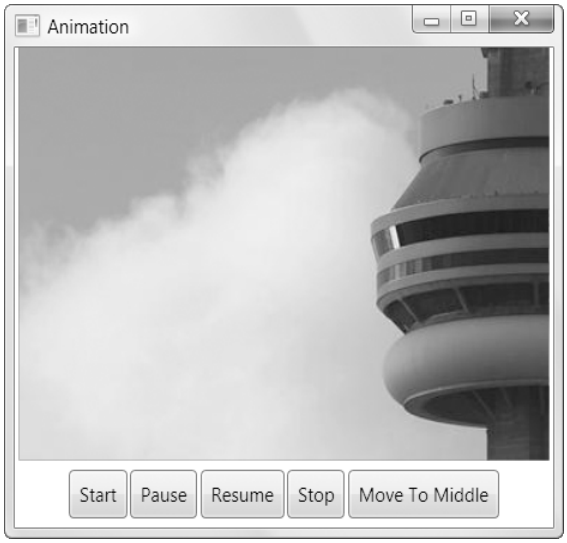

Figure 15-3. *A controllable animation*

Here's the markup that defines the Grid with its two images:

```
<Grid>
  <Image Source="night.jpg"></Image>
  <Image Source="day.jpg" Name="imgDay"></Image>
</Grid>
```

and here's the animation that fades from one to the other:

```
<DoubleAnimation
  Storyboard.TargetName="imgDay" Storyboard.TargetProperty="Opacity"
  From="1" To="0" Duration="0:0:10">
</DoubleAnimation>
```

To make this example more interesting, it includes several buttons at the bottom that allow you to control the playback of this animation. Using these buttons, you can perform the typical media player actions, such as pausing, resuming, and stopping. (You could add other buttons to change the speed ratio and seek out specific times.)

Here's the markup that defines these buttons:

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center" Margin="5">
  <Button Name="cmdStart">Start</Button>
  <Button Name="cmdPause">Pause</Button>
  <Button Name="cmdResume">Resume</Button>
  <Button Name="cmdStop">Stop</Button>
  <Button Name="cmdMiddle">Move To Middle</Button>
</StackPanel>
```

Ordinarily, you might choose to place the event trigger in the Triggers collection of each individual button. However, as explained earlier, that doesn't work for animations. The easiest solution is to define all the event triggers in one place, such as the Triggers collection of a containing element, and wire them up using the EventTrigger.SourceName property. As long as the SourceName matches the Name property you've given the button, the trigger will be applied to the appropriate button.

In this example, you could use the Triggers collection of the StackPanel that holds the buttons. However, it's often easier to use the Triggers collection of the top-level element, which is the window in this case. That way, you can move your buttons to different places in your user interface without disabling their functionality.

```
<Window.Triggers>
  <EventTrigger SourceName="cmdStart" RoutedEvent="Button.Click">
    <BeginStoryboard Name="fadeStoryboardBegin">
      <Storyboard>
        <DoubleAnimation
          Storyboard.TargetName="imgDay" Storyboard.TargetProperty="Opacity"
          From="1" To="0" Duration="0:0:10">
        </DoubleAnimation>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>

  <EventTrigger SourceName="cmdPause" RoutedEvent="Button.Click">
    <PauseStoryboard BeginStoryboardName="fadeStoryboardBegin"></PauseStoryboard>
  </EventTrigger>
  <EventTrigger SourceName="cmdResume" RoutedEvent="Button.Click">
    <ResumeStoryboard BeginStoryboardName="fadeStoryboardBegin"></ResumeStoryboard>
  </EventTrigger>
  <EventTrigger SourceName="cmdStop" RoutedEvent="Button.Click">
    <StopStoryboard BeginStoryboardName="fadeStoryboardBegin"></StopStoryboard>
```

```
  </EventTrigger>
  <EventTrigger SourceName="cmdMiddle" RoutedEvent="Button.Click">
    <SeekStoryboard BeginStoryboardName="fadeStoryboardBegin"
     Offset="0:0:5"></SeekStoryboard>
  </EventTrigger>
</Window.Triggers>
```

Notice that you must give a name to the BeginStoryboard action. (In this example, it's fadeStoryboardBegin). The other triggers specify this name in the BeginStoryboardName property to link up to the same storyboard.

You'll encounter one limitation when using storyboard actions. The properties they provide (such as SeekStoryboard.Offset and SetStoryboardSpeedRatio.SpeedRatio) are not dependency properties. That limits your ability to use data binding expressions. For example, you can't automatically read the Slider.Value property and apply it to the SetStoryboardSpeedRatio.SpeedRatio action, because the SpeedRatio property doesn't accept a data binding expression. You might think you could code around this problem by using the SpeedRatio property of the Storyboard object, but this won't work. When the animation starts, the SpeedRatio value is read and used to create an animation clock. If you change it after that point, the animation continues at its normal pace.

If you want to adjust the speed or position dynamically, the only solution is to use code. The Storyboard class exposes methods that provide the same functionality as the triggers described in Table 15-2, including Begin(), Pause(), Resume(), Seek(), Stop(), SkipToFill(), SetSpeedRatio(), and Remove().

To access the Storyboard object, you need to make sure you set its Name property in the markup:

```
<Storyboard Name="fadeStoryboard">
```

■ **Note** Don't confuse the name of the Storyboard object (which is required to use the storyboard in your code) with the name of the BeginStoryboard action (which is required to wire up other trigger actions that manipulate the storyboard). To prevent confusion, you may want to adopt a convention like adding the word *Begin* to the end of the BeginStoryboard name.

Now you simply need to write the appropriate event handler and use the methods of the Storyboard object. (Remember, simply changing storyboard properties such as SpeedRatio won't have any effect. They simply configure the settings that will be used when the animation starts.)

Here's an event handler that reacts when you drag the thumb on a Slider. It then takes the value of the slider (which ranges from 0 to 3) and uses it to apply a new speed ratio:

```
private void sldSpeed_ValueChanged(object sender, RoutedEventArgs e)
{
    fadeStoryboard.SetSpeedRatio(this, sldSpeed.Value);
}
```

Notice that the SetSpeedRatio() requires two arguments. The first argument is the top-level animation container (in this case, the current window). All the storyboard methods require this reference. The second argument is the new speed ratio.

The Wipe Effect

The previous example provides a gradual transition between the two images you're using by varying the Opacity of the topmost image. Another common way to transition between images is to perform a "wipe" that unveils the new image overtop the existing one.

The trick to using this technique is to create an opacity mask for the topmost image. Here's an example:

```
<Image Source="day.jpg" Name="imgDay">
  <Image.OpacityMask>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
      <GradientStop Offset="0" Color="Transparent" x:Name="transparentStop" />
      <GradientStop Offset="0" Color="Black" x:Name="visibleStop" />
    </LinearGradientBrush>
  </Image.OpacityMask>
</Image>
```

This opacity mask uses a gradient that defines two gradient stops, Black (where the image will be completely visible) and Transparent (where the image will be completely transparent). Initially, both stops are positioned at the left edge of the image. Because the visible stop is declared last, it takes precedence, and the image will be completely opaque. Notice that both stops are named so they can be easily accessed by your animation.

Next, you need to perform your animation on the offsets of the LinearGradientBrush. In this example, both offsets are moved from the left side to the right side, allowing the image underneath to appear. To make this example a bit fancier, the offsets don't occupy the same position while they move. Instead, the visible offset leads the way, followed by the transparent offset after a short delay of 0.2 seconds. This creates a blended fringe at the edge of the wipe while the animation is underway.

```
<Storyboard>
  <DoubleAnimation
    Storyboard.TargetName="visibleStop"
    Storyboard.TargetProperty="Offset"
    From="0" To="1.2" Duration="0:0:1.2" ></DoubleAnimation>
  <DoubleAnimation
    Storyboard.TargetName="transparentStop"
    Storyboard.TargetProperty="Offset" BeginTime="0:0:0.2"
    From="0" To="1" Duration="0:0:1" ></DoubleAnimation>
</Storyboard>
```

There's one odd detail here. The visible stop moves to 1.2 rather than simply 1, which denotes the right edge of the image. This ensures that both offsets move at the same speed, because the total distance each one must cover is proportional to the duration of its animation.

Wipes commonly work from left to right or top to bottom, but more creative effects are possible by using different opacity masks. For example, you could use a DrawingBrush for your opacity mask and modify its geometry to let the content underneath show through in a tiled pattern. You'll see more examples that animate brushes in Chapter 16.

Monitoring Progress

The animation player shown in Figure 15-3 still lacks one feature that's common in most media players—the ability to determine your current position. To make it a bit fancier, you can add some text that shows the time offset and a progress bar that provides a visual indication of how far you are in the animation. Figure 15-4 shows a revised animation player with both details (along with the Slider for controlling speed that was explained in the previous section).

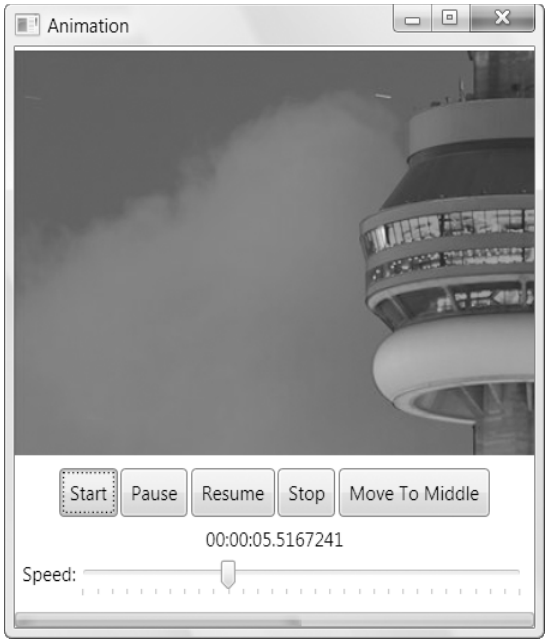

Figure 15-4. *Displaying position and progress in an animation*

Adding these details is fairly straightforward. First you need a TextBlock element to show the time and a ProgressBar control to show the graphical bar. You might assume you could set the TextBlock value and the ProgressBar content using a data binding expression, but this isn't possible. That's because the only way to retrieve the information about the current animation clock from the Storyboard is to use methods such as GetCurrentTime() and GetCurrentProgress(). There isn't any way to get the same information from properties.

The easiest solution is to react to one of the storyboard events listed in Table 15-3.

Table 15-3. *Storyboard Events*

Name	Description
Completed	The animation has reached its ending point.
CurrentGlobalSpeedInvalidated	The speed has changed, or the animation has been paused, resumed, stopped, or moved to a new position. This event also occurs when the animation clock reverses (at the end of a reversible animation) and when it accelerates or decelerates.
CurrentStateInvalidated	The animation has started or ended.
CurrentTimeInvalidated	The animation clock has moved forward an increment, changing the animation. This event also occurs when the animation starts, stops, or ends.
RemoveRequested	The animation is being removed. The animated property will subsequently return to its original value.

In this case, the event you need is CurrentTimeInvalidated, which fires every time the animation clock moves forward. (Typically, this will be 60 times per second, but if your code takes more time to execute, you may miss clock ticks.)

When the CurrentTimeInvalidated event fires, the sender is a Clock object (from the System.Windows.Media.Animation namespace). The Clock object allows you to retrieve the current time as a TimeSpan and the current progress as a value from 0 to 1.

Here's the code that updates the label and the progress bar:

```
private void storyboard_CurrentTimeInvalidated(object sender, EventArgs e)
{
    Clock storyboardClock = (Clock)sender;

    if (storyboardClock.CurrentProgress == null)
    {
        lblTime.Text = "[[ stopped ]]";
        progressBar.Value = 0;
    }
    else
    {
        lblTime.Text = storyboardClock.CurrentTime.ToString();
        progressBar.Value = (double)storyboardClock.CurrentProgress;
    }
}
```

■ **Tip** If you use the Clock.CurrentProgress property, you don't need to perform any calculation to determine the value for your progress bar. Instead, simply configure your progress bar with a minimum of 0 and a maximum of 1. That way, you can simply use the Clock.CurrentProgress to set the ProgressBar.Value, as in this example.

Animation Easing

One of the shortcomings of linear animation is that it often feels mechanical and unnatural. By comparison, sophisticated user interfaces have animated effects that model real-world systems. For example, they may use tactile push-buttons that jump back quickly when clicked but slow down as they come to rest, creating the illusion of true movement. Or, they may use maximize and minimize effects like the Windows operating system, where the speed at which the window grows or shrinks accelerates as the window nears its final size. These details are subtle, and you're not likely to notice them when they're implemented well. However, you'll almost certainly notice the clumsy feeling of less refined animations that lack these finer points.

The secret to improving your animations and creating more natural animations is to vary the rate of change. Instead of creating animations that change properties at a fixed, unchanging rate, you need to design animations that speed up or slow down along the way. WPF gives you several options. In the next chapter, you'll learn about frame-based animation and key frame animation, two techniques that give you more nuanced control over your animations (and require significantly more work). But the simplest way to make a more natural animation is to use a prebuilt *easing function*.

When using an easing function, you still define your animation normally by specifying the starting and ending property values. But in addition to these details, you add a ready-made mathematical function that alters the progression of your animation, causing it to accelerate or decelerate at different points. This is the technique you'll study in the following sections.

Using an Easing Function

The best part about animation easing is that it requires much less work than other approaches such as frame-based animation and key frames. To use animation easing, you set the EasingFunction property of an animation object with an instance of an easing function class (a class that derives from EasingFunctionBase). You'll usually need to set a few properties on the easing function, and you may be forced to play around with different settings to get the effect you want, but you'll need no code and very little additional XAML.

For example, consider the two animations shown here, which act on a button. When the user moves the mouse over the button, a small snippet of code calls the growStoryboard animation into action, stretching the button to 400 units. When the user moves the mouse off the button, the buttons shrinks back to its normal size.

```
<Storyboard x:Name="growStoryboard">
  <DoubleAnimation
   Storyboard.TargetName="cmdGrow" Storyboard.TargetProperty="Width"
   To="400" Duration="0:0:1.5"></DoubleAnimation>
</Storyboard>
```