MCTS Self-Paced Training Kit (Exam 70-502): Microsoft® .NET Framework 3.5—Windows® Presentation Foundation

Matthew A. Stoecker

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books12485.aspx

9780735625662

Microsoft Press

Table of Contents

7 Styles and Animation 303

Chapter 7

Styles and Animation

One of the major advances with the advent of the Windows Presentation Foundation (WPF) programming model is the uniquely agile use of the system's visual capabilities. In this chapter, you learn to use two aspects of WPF programming that take full advantage of these capabilities: styles and animation. Styles allow you to quickly apply changes to the visual interface and change the look and feel of your application in response to different conditions. Animations allow you to change property values over timelines that can be useful for a variety of visual effects. Together, these features allow you to harness the full power of the WPF presentation layer.

Exam objectives in this chapter:

- Create a consistent user interface appearance by using styles.
- Change the appearance of a UI element by using triggers.
- Add interactivity by using animations.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book
- Microsoft Visual Studio 2008 Professional Edition installed on your computer
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with Microsoft .NET Framework version 3.5
- An understanding of Extensible Application Markup Language (XAML)

Real World

Matthew Stoecker

At every turn, it seems that WPF provides more and more support for the creation of rich visual interfaces. The support for styles and triggers enables the rapid creation of interactive visual interfaces that used to take hours of coding event handlers. Likewise, animations now open up the possibilities for stunning user interfaces with minimal effort. I'm glad that now I can create attractive applications with the same amount of effort that the boxy old Windows Forms apps took!

Lesson 1: Styles

Styles allow you to create a cohesive look and feel for your application. You can use styles to define a standard color and sizing scheme for your application and use triggers to provide dynamic interaction with your UI elements. In this lesson, you learn to create and implement styles. You learn to apply a style to all instances of a single type and to implement style inheritance. You learn to use setters to set properties and event handlers, and you learn to use triggers to change property values dynamically. Finally, you learn about the order of property precedence.

After this lesson, you will be able to:

- Create and implement a style
- Apply a style to all instances of a type
- Implement style inheritance
- Use property and event setters
- Explain the order of property value precedence
- Use and implement triggers, including property triggers, data triggers, event triggers, and multiple triggers

Estimated lesson time: 30 minutes

Using Styles

Styles can be thought of as analogous to cascading style sheets as used in Hypertext Markup Language (HTML) pages. Styles basically tell the presentation layer to substitute a new visual appearance for the standard one. They allow you to make changes to the user interface as a whole easily and to provide a consistent look and feel for your application in a variety of situations. Styles enable you to set properties and hook up events on UI elements through the application of those styles. Further, you can create visual elements that respond dynamically to property changes through the application of triggers, which listen for a property change and then apply style changes in response.

Properties of Styles

The primary class in the application of styles is, unsurprisingly, the *Style* class. The *Style* class contains information about styling a group of properties. A *Style* can be created to apply to a single instance of an element, to all instances of an element type, or across multiple types. The important properties of the *Style* class are shown in Table 7-1.

Table 7-1 Important Properties of the *Style* Class

Property	Description
BasedOn	Indicates another style that this style is based on. This property is useful for creating inherited styles.
Resources	Contains a collection of local resources used by the style. The *Resources* property is discussed in detail in Chapter 9, "Resources, Documents, and Localization."
Setters	Contains a collection of *Setter* or *EventSetter* objects. These are used to set properties or events on an element as part of a style.
TargetType	This property identifies the intended element type for the style.
Triggers	Contains a collection of *Trigger* objects and related objects that allow you to designate a change in the user interface in response to changes in properties.

The basic skeleton of a <*Style*> element in XAML markup looks like the following:

```
<Style>
   <!-- A collection of setters is enumerated here -->
   <Style.Triggers>
   <!-- A collection of Trigger and related objects is enumerated here -->
   </Style.Triggers>
   <Style.Resources>
      <!-- A collection of local resources for use in the style -->
   </Style.Resources>
</Style>
```

Setters

The most common class you will use in the construction of Styles is the *Setter*. As their name implies, *Setters* are responsible for setting some aspect of an element. *Setters* come in two flavors: property setters (or just *Setters*, as they are called in markup), which set values for properties; and event setters, which set handlers for events.

Property Setters

Property setters, represented by the <*Setter*> tag in XAML, allow you to set properties of elements to specific values. A property setter has two important properties: the *Property* property, which designates the property that is to be set by the *Setter,* and the *Value* property, which indicates the value to which the property is to be set. The following example demonstrates a *Setter* that sets the Background property of a *Button* element to *Red:*

```
<Setter Property="Button.Background" Value="Red" />
```

The value for the *Property* property must take the form of the following:

```
Element.PropertyName
```

If you want to create a style that sets a property on multiple different types of elements, you could set the style on a common class that the elements inherit, as shown here:

```
<Style>
   <Setter Property="Control.Background" Value="Red" />
</Style>
```

This style sets the *Background* property of all elements that inherit from the *Control* to which it is applied.

Event Setters

Event setters (represented by the <*EventSetter*> tag) are similar to property setters, but they set event handlers rather than property values. The two important properties for an *EventSetter* are the *Event* property, which specifies the event for which the handler is being set; and the *Handler* property, which specifies the event handler to attach to that event. An example is shown here:

```
<EventSetter Event="Button.MouseEnter" Handler="Button_MouseEnter" />
```

The value of the *Handler* property must specify an extant event handler with the correct signature for the type of event with which it is connected. Similar to property setters, the format for the *Event* property is

```
Element.EventName
```

where the element type is specified, followed by the event name.

Creating a Style

You’ve seen the simplest possible implementation of a style: a single *Setter* between two *Style* tags, but you haven’t yet seen how to apply a style to an element. There are several ways to apply a style to an element or elements. This section examines the various ways to apply a style to elements in your user interface.

Setting the *Style* Property Directly

The most straightforward way to apply a style to an element is to set the *Style* property directly in XAML. The following example demonstrates directly setting the *Style* property of a *Button* element:

```
<Button Height="25" Name="Button1" Width="100">
   <Button.Style>
      <Style>
         <Setter Property="Button.Content" Value="Style set directly" />
         <Setter Property="Button.Background" Value="Red" />
      </Style>
   </Button.Style>
</Button>
```

While setting the *Style* directly in an element might be the most straightforward, it is seldom the best method. When setting the *Style* directly, you must set it for each element that you want to be affected by the *Style*. In most cases, it is simpler to set the properties of the element directly at design time.

One scenario where you might want to set the *Style* directly on an element is to provide a set of *Triggers* for that element. Because *Triggers* must be set in a *Style* (except for *EventTriggers*, as you will see in the next section), you conceivably could set the *Style* directly to set triggers for an element.

Setting a Style in a *Resources* Collection

The most common method for setting styles is to create the style as a member of a *Resources* collection and then apply the style to elements in your user interface by referencing the resource. The following example demonstrates creating a style as part of the *Windows.Resources* collection:

```
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style defined in resources" />
      <Setter Property="Button.Background" Value="Red" />
   </Style>
</Window.Resources>
```

Under most circumstances, you must supply a key value for a *Style* that you define in the *Resources* collection. Then you can apply that style to an element by referencing the resource, as shown in bold here:

```
<Button Name="Button1" Style="{StaticResource StyleOne}" Height="30"
   Width="200" />
```

The advantage to defining a *Style* in the *Resources* section is that you can then apply that *Style* to multiple elements by simply referencing the resource. Resources are discussed in detail in Chapter 9.

Applying Styles to All Controls of a Specific Type

You can use the *TargetType* property to specify a type of element to be associated with the style. When you set the *TargetType* property on a *Style,* that *Style* is applied to all elements of that type automatically. Further, you do not need to specify the qualifying type name in the *Property* property of any *Setters* that you use—you can just refer to the property name. When you specify the *TargetType* for a *Style* that you have defined in a *Resources* collection, you do not need to provide a key value for that style. The following example demonstrates the use of the *TargetType* property:

```
<Window.Resources>
   <Style TargetType="Button">
      <Setter Property=" Content" Value="Style set for all buttons" />
      <Setter Property="Background" Value="Red" />
   </Style>
</Window.Resources>
```

When you apply the *TargetType* property, you do not need to add any additional markup to the elements of that type to apply the style.

If you want an individual element to opt out of the style, you can set the style on that element explicitly, as seen here:

```
<Button Style="{x:Null}" Margin="10">No Style</Button>
```

This example explicitly sets the *Style* to *Null,* which causes the *Button* to revert to its default look. You also can set the *Style* to another *Style* directly, as seen earlier in this lesson.

Setting a Style Programmatically

You can create and define a style programmatically. While defining styles in XAML is usually the best choice, creating a style programmatically might be useful when you want to create and apply a new style dynamically, possibly based on user preferences.

The typical method for creating a style programmatically is to create the *Style* object in code; then create *Setters* (and *Triggers*, if appropriate); add them to the appropriate collection on the *Style* object; and then when finished, set the *Style* property on the target elements. The following example demonstrates creating and applying a simple style in code:

```
' VB
Dim aStyle As New Style
Dim aSetter As New Setter
aSetter.Property = Button.BackgroundProperty
aSetter.Value = Brushes.Red
aStyle.Setters.Add(aSetter)
Dim bSetter As New Setter
bSetter.Property = Button.ContentProperty
bSetter.Value = "Style set programmatically"
aStyle.Setters.Add(bSetter)
Button1.Style = aStyle

// C#
Style aStyle = new Style();
Setter aSetter = new Setter();
aSetter.Property = Button.BackgroundProperty;
aSetter.Value = Brushes.Red;
aStyle.Setters.Add(aSetter);
Setter bSetter = new Setter();
bSetter.Property = Button.ContentProperty;
bSetter.Value = "Style set programmatically";
aStyle.Setters.Add(bSetter);
Button1.Style = aStyle;
```

You can also define a style in a *Resources* collection and apply that style in code, as shown here:

```
<!-- XAML -->
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style applied in code" />
      <Setter Property="Button.Background" Value="Red" />
   </Style>
</Window.Resources>

' VB
Dim aStyle As Style
aStyle = CType(Me.Resources("StyleOne"), Style)
Button1.Style = aStyle

// C#
Style aStyle;
aStyle = (Style)this.Resources["StyleOne"];
Button1.Style = aStyle;
```

Implementing Style Inheritance

You can use inheritance to create styles that conform to the basic look and feel of the original style but provide differences that offset some controls from others. For example, you might create one *Style* for all the *Button* elements in your user interface and create an inherited style to provide emphasis for one of the buttons. You can use the *BasedOn* property to create *Style* objects that inherit from other *Style* objects. The *BasedOn* property references another style and automatically inherits all the members of that *Style* and then allows you to build on that *Style* by adding additional members. The following example demonstrates two *Style* objects—an original *Style* and a *Style* that inherits it:

```
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style set in original Style" />
      <Setter Property="Button.Background" Value="Red" />
      <Setter Property="Button.FontSize" Value="15" />
      <Setter Property="Button.FontFamily" Value="Arial" />
   </Style>
   <Style x:Key="StyleTwo" BasedOn="{StaticResource StyleOne}">
      <Setter Property="Button.Content" Value="Style set by inherited style" />
      <Setter Property="Button.Background" Value="AliceBlue" />
      <Setter Property="Button.FontStyle" Value="Italic" />
   </Style>
</Window.Resources>
```

The result of applying these two styles is seen in Figure 7-1.

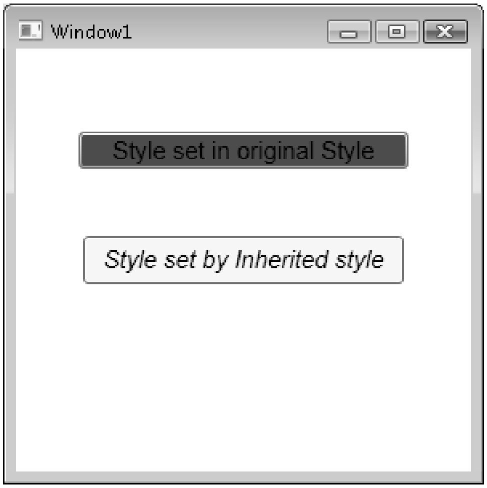

Figure 7-1 Two buttons—the original and an inherited style

When a property is set in both the original style and the inherited style, the property value set by the inherited style always takes precedence. But when a property is set by the original style and not set by the inherited style, the original property setting is retained.

Quick Check

- Under what circumstances is a *Style* automatically applied to an element? How else can a *Style* be applied to an element?

Quick Check Answer

- A *Style* is applied to an element automatically when it is declared as a resource in the page and the *TargetType* property of the *Style* is set. If the *TargetType* property is not set, you can apply a *Style* to an element by setting that element's *Style* property, either in XAML or in code.

Triggers

Along with *Setters, Triggers* make up the bulk of objects that you use in creating styles. *Triggers* allow you to implement property changes declaratively in response to other property changes that would have required event-handling code in Windows Forms programming. There are five kinds of *Trigger* objects, as listed in Table 7-2.

Table 7-2 Types of *Trigger* Objects

Type	Class Name	Description
Property trigger	*Trigger*	Monitors a property and activates when the value of that property matches the *Value* property.
Multi-trigger	*MultiTrigger*	Monitors multiple properties and activates only when all the monitored property values match their corresponding *Value* properties.
Data trigger	*DataTrigger*	Monitors a bound property and activates when the value of the bound property matches the *Value* property.

Table 7-2 Types of *Trigger* Objects

Type	Class Name	Description
Multi-data-trigger	*MultDataTrigger*	Monitors multiple bound properties and activates only when all the monitored bound properties match their corresponding *Value* properties.
Event trigger	*EventTrigger*	Initiates a series of *Actions* when a specified event is raised.

A *Trigger* is active only when it is part of a *Style.Triggers* collection—with one exception. *EventTrigger* objects can be created within a *Control.Triggers* collection outside a *Style*. The *Control.Triggers* collection can accommodate only *EventTriggers*, and any other *Trigger* placed in this collection causes an error. *EventTriggers* are primarily used with animation and are discussed further in Lesson 2 of this chapter, "Animations."

Property Triggers

The most commonly used type of *Trigger* is the property trigger. The property trigger monitors the value of a property specified by the *Property* property. When the value of the specified property equals the *Value* property, the *Trigger* is activated. Important properties of property triggers are shown in Table 7-3.

Table 7-3 Important Properties of Property Triggers

Property	Description
EnterActions	Contains a collection of *Action* objects that are applied when the *Trigger* becomes active. Actions are discussed in greater detail in Lesson 2 of this chapter.
ExitActions	Contains a collection of *Action* objects that are applied when the *Trigger* becomes inactive. *Actions* are discussed in greater detail in Lesson 2 of this chapter.
Property	Indicates the property that is monitored for changes.
Setters	Contains a collection of *Setter* objects that are applied when the *Trigger* becomes active.
Value	Indicates the value that is compared to the property referenced by the *Property* property.

Triggers listen to the property indicated by the *Property* property and compare that property to the *Value* property. When the referenced property and the *Value* property are equal, the *Trigger* is activated. Any *Setter* objects in the *Setters* collection of the *Trigger* are applied to the style, and any *Actions* in the *EnterActions* collections are initiated. When the referenced property no longer matches the *Value* property, the *Trigger* is inactivated. All *Setter* objects in the *Setters* collection of the *Trigger* are inactivated, and any *Actions* in the *ExitActions* collection are initiated.

NOTE *Actions* are used primarily in animations, and they are discussed in greater detail in Lesson 2 of this chapter.

The following example demonstrates a simple *Trigger* object that changes the *FontWeight* of a *Button* element to *Bold* when the mouse enters the *Button:*

```
<Style.Triggers>
   <Trigger Property="Button.IsMouseOver" Value="True">
      <Setter Property="Button.FontWeight" Value="Bold" />
   </Trigger>
</Style.Triggers>
```

In this example, the *Trigger* defines one *Setter* in its *Setters* collection. When the *Trigger* is activated, that *Setter* is applied.

Multi-triggers

Multi-triggers are similar to property triggers in that they monitor the value of properties and activate when those properties meet a specified value. The difference is that multi-triggers are capable of monitoring several properties at a single time and they activate only when all monitored properties equal their corresponding *Value* properties. The properties that are monitored and their corresponding *Value* properties are defined by a collection of *Condition* objects. The following example demonstrates a *MultiTrigger* that sets the *Button.FontWeight* property to *Bold* only when the *Button* is focused and the mouse has entered the control:

```
<Style.Triggers>
   <MultiTrigger>
      <MultiTrigger.Conditions>
         <Condition Property="Button.IsMouseOver" Value="True" />
         <Condition Property="Button.IsFocused" Value="True" />
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
         <Setter Property="Button.FontWeight" Value="Bold" />
      </MultiTrigger.Setters>
   </MultiTrigger>
</Style.Triggers>
```

Data Triggers and Multi-data-triggers

Data triggers are similar to property triggers in that they monitor a property and activate when the property meets a specified value, but they differ in that the property they monitor is a bound property. Instead of a *Property* property, data triggers expose a *Binding* property that indicates the bound property to listen to. The following shows a data trigger that changes the *Background* property of a *Label* to *Red* when the bound property *CustomerName* equals "Fabrikam":

```
<Style.Triggers>
   <DataTrigger Binding="{Binding Path=CustomerName}" Value="Fabrikam">
      <Setter Property="Label.Background" Value="Red" />
   </DataTrigger>
</Style.Triggers>
```

Multi-data-triggers are to data triggers as multi-triggers are to property triggers. They contain a collection of *Condition* objects, each of which specifies a bound property via its *Binding* property and a value to compare to that bound property. When all the conditions are satisfied, the *MultiDataTrigger* activates. The following example demonstrates a *MultiDataTrigger* that sets the *Label.Background* property to *Red* when *CustomerName* equals "Fabrikam" and *OrderSize* equals 500:

```
<Style.Triggers>
   <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
         <Condition Binding="{Binding Path=CustomerName}" Value="Fabrikam" />
         <Condition Binding="{Binding Path=OrderSize}" Value="500" />
      </MultiDataTrigger.Conditions>
      <MultiDataTrigger.Setters>
         <Setter Property="Label.Background" Value="Red" />
      </MultiDataTrigger.Setters>
   </MultiDataTrigger>
</Style.Triggers>
```

Event Triggers

Event triggers are different from the other *Trigger* types. While other *Trigger* types monitor the value of a property and compare it to an indicated value, event triggers specify an event and activate when that event is raised. In addition, event triggers do not have a *Setters* collection—rather, they have an *Actions* collection. Although you have been exposed briefly to the *SoundPlayerAction* in Chapter 4, "Adding and Managing Content," most actions deal with animations, which are discussed in detail in Lesson 2 of this chapter. The following two examples demonstrate the *EventTrigger* class. The first example uses a *SoundPlayerAction* to play a sound when a *Button* is clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <SoundPlayerAction Source="C:\myFile.wav" />
</EventTrigger>
```

The second example demonstrates a simple animation that causes the *Button* to grow in height by 200 units when clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <EventTrigger.Actions>
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation Duration="0:0:5"
               Storyboard.TargetProperty="Height" To="200" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger.Actions>
</EventTrigger>
```

Understanding Property Value Precedence

By now, you have probably noticed that a property can be set in many different ways. They can be set in code; they can be set by styles; they can have default values; and so on. It might seem logical at first to believe that a property will have the value it was last set to, but this is actually incorrect. There is a defined and strict order of precedence that determines a property's value based on *how* it was set, not when. The precedence order is summarized here, with highest precedence listed first:

1. Set by coercion by the property system.
2. Set by active animations or held animations.
3. Set locally, either by code, by direct setting in XAML, or through data binding.
4. Set by the *TemplatedParent*. Within this category, there is a sub-order of precedence, again listed in descending order:
   a. Set by *Triggers* from the templated parent
   b. Set by the templated parent through property sets
5. Implicit style—this applies only to the *Style* property.
6. Set by *Style* triggers.
7. Set by *Template* triggers.

8. Set by *Style* setters.
9. Set by the default *Style*. There is a sub-order within this category, again listed in descending order:
    a. Set by *Triggers* in the default style
    b. Set by *Setters* in the default style
10. Set by inheritance.
11. Set by metadata.

Exam Tip The order of property precedence seems complicated, but actually it is fairly logical. Be sure that you understand the concept behind the property order in addition to knowing the order itself.

This may seem like a complicated and arbitrary order of precedence, but upon closer examination it is actually very logical and based upon the needs of the application and the user. The highest precedence is property coercion. This takes place in some elements if an attempt is made to set a property beyond its allowed values. For example, if an attempt is made to set the *Value* property of a *Slider* control to a value higher than the *Maximum* property, the *Value* is coerced to equal the *Maximum* property. Next in precedence come animations. For animations to have any meaningful use, they must be able to override preset property values. The next highest level of precedence is properties that have been set explicitly through developer or user action.

Properties set by the *TemplatedParent* are next in the order of precedence. These are properties set on objects that come into being through a template. Templates are discussed further in Chapter 8, “Customizing the User Interface.” After this comes a special precedence item that applies only to the *Style* property of an element: Provided that the *Style* property has not been set by any item with a higher-level precedence, it is set to a *Style* whose *TargetType* property matches the type of the element in question. Then come properties set by *Triggers*—first those set by a *Style,* then those set by a *Template.* This is logical because for triggers to have any meaningful effect, they must override properties set by styles.

Properties set by styles come next: first properties set by user-defined styles, and then properties set by the default style (also called the *Theme,* which typically is set by the operating system). Finally come properties that are set through inheritance and the application of metadata.

For developers, there are a few important implications that are not intuitively obvious. The most important is that if you set a property explicitly—whether in XAML or in code—the explicitly set property blocks any changes dictated by a *Style* or *Trigger*. WPF assumes that you want that property value to be there for a reason and does not allow it to be set by a *Style* or *Trigger,* although it still can be overridden by an active animation.

A second, less obvious implication is that when using the Visual Studio designer to drag and drop items onto the design surface from the *ToolBox,* the designer explicitly sets several properties, especially layout properties. These property settings have the same precedence as they would if you had set them yourself. So if you are designing a style-oriented user interface, you should either enter XAML code directly in XAML view to create controls and set as few properties explicitly as possible, or you should review the XAML that Visual Studio generates and delete settings as appropriate.

You can clear a property value that has been set in XAML or code manually by calling the *DependencyObject.ClearValue* method. The following code example demonstrates how to clear the value of the *Width* property on a button named *Button1*:

```
' VB
Button1.ClearValue(WidthProperty)

// C#
Button1.ClearValue(WidthProperty);
```

Once the value has been cleared, it can be reset automatically by the property system.

Lab: Creating High-Contrast Styles

In this lab, you create a rudimentary high-contrast *Style* for *Button, TextBox,* and *Label* elements.

Exercise 1: Using Styles to Create High-Contrast Elements

1. Create a new WPF application in Visual Studio.
2. In XAML view, just above the *<Grid>* declaration, create a *Window.Resources* section, as shown here:

```
<Window.Resources>

</Window.Resources>
```

3. In the *Window.Resources* section, create a high-contrast *Style* for *TextBox* controls that sets the background color to *Black* and the foreground to *White*. The *TextBox* controls also should be slightly larger by default. An example is shown here:

```
<Style TargetType="TextBox">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="BorderBrush" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="30" />
</Style>
```

4. Create similar styles for *Button* and *Label,* as shown here:

```
<Style TargetType="Label">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="33" />
</Style>
<Style TargetType="Button">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="30" />
</Style>
```

5. Type the following in XAML view. Note that you should not add controls from the toolbox because that automatically sets some properties in the designer at a higher property precedence than styles:

```
<Label Margin="26,62,126,0" VerticalAlignment="Top">
   High-Contrast Label</Label>
<TextBox Margin="26,117,126,115">High-Contrast TextBox
   </TextBox>
<Button Margin="26,0,126,62" VerticalAlignment="Bottom">
   High-Contrast Button</Button>
```

6. Press F5 to build and run your application. Note that while the behavior of these controls is unaltered, their appearance has changed.

Exercise 2: Using Triggers to Enhance Visibility

1. In XAML view for the solution you completed in Exercise 1, add a *Style.Triggers* section to the *TextBox Style,* as shown here:

```
<Style.Triggers>

</Style.Triggers>
```

2. In the *Style.Triggers* section, add *Triggers* that detect when the mouse is over the control and enlarge the *FontSize* of the control, as shown here:

```
<Trigger Property="IsMouseOver" Value="True">
   <Setter Property="FontSize" Value="20" />
</Trigger>
```

3. Add similar *Style.Triggers* collections to your other two styles.
4. Press F5 to build and run your application. The *FontSize* of a control now increases when you move the mouse over it.

Lesson Summary

- *Styles* allow you to define consistent visual styles for your application. *Styles* use a collection of *Setters* to apply style changes. The most commonly used *Setter* type is the property setter, which allows you to set a property. Event setters allow you to hook up event handlers as part of an applied style.
- *Styles* can be set inline, but more frequently, they are defined in a *Resources* collection and are set by referring to the resource. You can apply a style to all instances of a control by setting the *TargetType* property to the appropriate type.
- *Styles* are most commonly applied declaratively, but they can be applied in code by creating a new style dynamically or obtaining a reference to a preexisting *Style* resource.
- You can create styles that inherit from other styles by using the *BasedOn* property.
- Property triggers monitor the value of a dependency property and can apply *Setters* from their *Setters* collection when the monitored property equals a predetermined value. Multi-triggers monitor multiple properties and apply their *Setters* when all monitored properties match corresponding specified values. Data triggers and multi-data-triggers are analogous but monitor bound values instead of dependency properties.
- Event triggers perform a set of *Actions* when a particular event is raised. They are used most commonly to control *Animations*.
- Property values follow a strict order of precedence depending on how they are set.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Styles." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Look at the following XAML snippet:

```
<Window.Resources>
   <Style x:Key="Style1">
      <Setter Property="Label.Background" Value="Blue" />
      <Setter Property="Button.Foreground" Value="Red" />
      <Setter Property="Button.Background" Value="LimeGreen" />
   </Style>
</Window.Resources>
<Grid>
   <Button Height="23" Margin="81,0,122,58" Name="Button1"
      VerticalAlignment="Bottom">Button</Button>
</Grid>
```

   Assuming that the developer hasn't set any properties any other way, what is the *Background* color of *Button1?*

   A. Blue

   B. Red

   C. LimeGreen

   D. System Default

2. Look at the following XAML snippet:

```
<Window.Resources>
   <Style x:Key="Style1">
      <Style.Triggers>
         <MultiTrigger>
            <MultiTrigger.Conditions>
               <Condition Property="TextBox.IsMouseOver"
                  Value="True" />
               <Condition Property="TextBox.IsFocused"
                  Value="True" />
            </MultiTrigger.Conditions>
            <Setter Property="TextBox.Background"
               Value="Red" />
         </MultiTrigger>
      </Style.Triggers>
   </Style>
</Window.Resources>
<Grid>
   <TextBox Style="{StaticResource Style1}" Height="21"
      Margin="75,0,83,108" Name="TextBox1"
      VerticalAlignment="Bottom" />
</Grid>
```

When will *TextBox1* appear with a red background?

- **A.** When the mouse is over *TextBox1*
- **B.** When *TextBox1* is focused
- **C.** When *TextBox1* is focused and the mouse is over *TextBox1*
- **D.** All of the above
- **E.** Never

3. Look at the following XAML snippet:

```
<Window.Resources>
   <Style TargetType="Button">
      <Setter Property="Content" Value="Hello" />
      <Style.Triggers>
         <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Content" Value="World" />
         </Trigger>
         <Trigger Property="IsMouseOver" Value="False">
            <Setter Property="Content" Value="How are you?" />
         </Trigger>
      </Style.Triggers>
   </Style>
</Window.Resources>
<Grid>
   <Button Height="23" Margin="81,0,122,58" Name="Button1"
      VerticalAlignment="Bottom">Button</Button>
</Grid>
```

What does *Button1* display when the mouse is NOT over the *Button?*

- **A.** *Hello*
- **B.** *World*
- **C.** *Button*
- **D.** *How are you?*

Lesson 2: Animations

Animations are another new feature of WPF. Animations allow you to change the value of a property over the course of a set period of time. Using this technique, you can create a variety of visual effects, including causing controls to grow or move abut the user interface, to change color gradually, or to change other properties over time. In this lesson, you learn how to create animations that animate a variety of property types and use *Storyboard* objects to control the playback of those animations.

After this lesson, you will be able to:

- Create and use animations
- Control animations with the *Storyboard* class
- Control timelines and playback of animations
- Implement simultaneous animations
- Use *Actions* to control animation playback
- Implement animations that use key frames
- Create and start animations in code

Estimated lesson time: 30 minutes

Using Animations

The term *animation* brings to mind hand-drawn anthropomorphic animals performing amusing antics in video media, but in WPF, animation has a far simpler meaning. Generally speaking, an animation in WPF refers to an automated property change over a set period of time. You can animate an element's size, location, color, or virtually any other property or properties associated with an element. You can use the *Animation* classes to implement these changes.

The *Animation* classes are a large group of classes designed to implement these automated property changes. There are 42 *Animation* classes in the *System .Windows.Media.Animation* namespace, and each one has a specific data type that they are designed to animate. *Animation* classes fall into three basic groups: Linear animations, key frame–based animations, and path-based animations.

Linear animations, which automate a property change in a linear way, are named in the format <*TypeName*>*Animation,* where <*TypeName*> is the name of the type being animated. *DoubleAnimation* is an example of a linear animation class, and that is the animation class you are likely to use the most.

Key frame–based animations perform their animation on the basis of several waypoints, called key frames. The flow of a key-frame animation starts at the beginning, and then progresses to each of the key frames before ending. The progression is usually linear. Key-frame animations are named in the format <*TypeName*>*AnimationUsingKeyFrames,* where <*TypeName*> is the name of the *Type* being animated. An example is *StringAnimationUsingKeyFrames.*

Path-based animations use a *Path* object to guide the animation. They are used most often to animate properties that relate to the movement of visual objects along a complex course. Path-based animations are named in the format <*TypeName*>*AnimationUsingPath,* where <*TypeName*> is the name of the type being animated. There are currently only three path-based *Animation* classes—*PointAnimationUsingPath, DoubleAnimationUsingPath,* and *MatrixAnimationUsingPath.*

Important Properties of Animations

Although there are many different *Animation* classes, they all work in the same fundamental way—they change the value of a designated property over a period of time. As such, they share common properties. Many of these properties also are shared with the *Storyboard* class, which is used to organize *Animation* objects, as you will see later in this lesson. Important common properties of the *Animation* and *Storyboard* classes are shown in Table 7-4.

Table 7-4 Important Properties of the *Animation* and *Storyboard* Classes

Property	Description
AccelerationRatio	Gets or sets a value specifying the percentage of the *Duration* property of the *Animation* that is spent accelerating the passage of time from zero to its maximum rate.
AutoReverse	Gets or sets a value that indicates whether the *Animation* plays in reverse after it completes a forward iteration.
BeginTime	Gets or sets the time at which the *Animation* should begin, relative to the time that the *Animation* is executed. For example, an *Animation* with a *BeginTime* set to 0:0:5 exhibits a 5-second delay before beginning.
DecelerationRatio	Gets or sets a value specifying the percentage of the duration of the *Animation* spent decelerating the passage of time from its maximum rate to zero.

Table 7-4 Important Properties of the *Animation* and *Storyboard* Classes

Property	Description
Duration	Gets or sets the length of time for which the *Animation* plays.
FillBehavior	Gets or sets a value that indicates how the *Animation* behaves after it has completed.
RepeatBehavior	Gets or sets a value that indicates how the *Animation* repeats.
SpeedRatio	Gets or sets the rate at which the *Animation* progresses relative to its parent.

In addition, the linear animation classes typically implement a few more important properties, which are described in Table 7-5.

Table 7-5 Important Properties of Linear Animation Classes

Property	Description
From	Gets or sets the starting value of the *Animation*. If omitted, the *Animation* uses the current property value.
To	Gets or sets the ending value of the *Animation*.
By	Gets or sets the amount by which to increase the value of the target property over the course of the *Animation*. If both the *To* and *By* properties are set, the value of the *By* property is ignored.

The following example demonstrates a very simple animation. This animation changes the value of a property that has a *Double* data type representation from 1 to 200 over the course of 10 seconds:

```
<DoubleAnimation Duration="0:0:10" From="1" To="200" />
```

In this example, the *Duration* property specifies a duration of 10 seconds for the animation, and the *From* and *To* properties indicate a starting value of 1 and an ending value of 200.

You might notice that something seems to be missing from this example. What property is this animation animating? The answer is that it is not animating any property—the *Animation* object carries no intrinsic information about the property that is being animated, but instead it is applied to a property by means of a *Storyboard*.

Storyboard Objects

The *Storyboard* is the object that controls and organizes animations in your user interface. The *Storyboard* class contains a *Children* collection, which organizes a collection of *Timeline* objects, which include *Animation* objects. When created declaratively in XAML, all *Animation* objects must be enclosed within a *Storyboard* object, as shown here:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200" />
</Storyboard>
```

Using a *Storyboard* to Control Animations

In XAML, *Storyboard* objects organize your *Animation* objects. The most important feature of the *Storyboard* object is that it contains properties that allow you to specify the target element and target property of the child *Animation* objects, as shown in bold in this example:

```
<Storyboard TargetName="Button1" TargetProperty="Height">
   <DoubleAnimation Duration="0:0:10" From="1" To="200" />
</Storyboard>
```

This example is now usable. It defines a timeline where over the course of 10 seconds, the *Height* property of *Button1* goes from a value of 1 to a value of 200.

The *TargetName* and *TargetProperty* properties are attached properties, so instead of defining them in the *Storyboard* itself, you can define them in the child *Animation* objects, as shown in bold here:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Height" />
</Storyboard>
```

Because a *Storyboard* can hold more than one *Animation* at a time, this configuration allows you to set separate target elements and properties for each animation. Thus, it is more common to use the attached properties.

Simultaneous Animations

The *Storyboard* can contain multiple child *Animation* objects. When the *Storyboard* is activated, all child animations are started at the same time and run simultaneously.

The following example demonstrates two simultaneous *Animations* that cause both the *Height* and *Width* of a *Button* element to grow over 10 seconds:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Height" />
   <DoubleAnimation Duration="0:0:10" From="1" To="100"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Widtht" />
</Storyboard>
```

Using *Animations* with *Triggers*

You now have learned most of the story about using *Animation* objects. The *Animation* object defines a property change over time, and the *Storyboard* object contains *Animation* objects and determines what element and property the *Animation* objects affect. But there is still one piece that is missing: How do you start and stop an *Animation?*

All declaratively created *Animation* objects must be housed within a *Trigger* object. This can be either as a part of a *Style,* or in the *Triggers* collection of an *Element,* which accepts only *EventTrigger* objects.

Trigger objects define collections of *Action* objects, which control when an *Animation* is started and stopped. The following example demonstrates an *EventTrigger* object with an inline *Animation:*

```
<EventTrigger RoutedEvent="Button.Click">
   <EventTrigger.Actions>
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation Duration="0:0:5"
               Storyboard.TargetProperty="Height" To="200" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger.Actions>
</EventTrigger>
```

As you can see in the preceding example, the *Storyboard* object is enclosed in a *BeginStoryboard* tag, which itself is enclosed in the *EventTrigger.Actions* tag. *BeginStoryboard* is an *Action*–it indicates that the contained *Storyboard* should be started. The *EventTrigger* class defines a collection of *Actions* that should be initiated when the *Trigger* is activated, and in this example, *BeginStoryboard* is the action that is initiated. Thus, when the *Button* indicated in this trigger is clicked, the described *Animation* runs.

Using *Actions* to Control Playback

There are several *Action* classes that can be used to manage animation playback. These classes are summarized in Table 7-6.

Table 7-6 Animation-Related *Action* Classes

Action	Description
BeginStoryboard	Begins the child *Storyboard* object.
PauseStoryboard	Pauses the playback of an indicated *Storyboard* at the current playback position.
ResumeStoryboard	Resumes playback of an indicated *Storyboard.*
SeekStoryboard	Fast-forwards to a specified position in a target *Storyboard.*
SetStoryboardSpeedRatio	Sets the *SpeedRatio* of the specified *Storyboard.*
SkipStoryboardToFill	Moves the specified *Storyboard* to the end of its timeline.
StopStoryboard	Stops playback of the specified *Storyboard* and returns the animation to the starting position.

PauseStoryboard, ResumeStoryboard, SkipStoryboardToFill, and *StopStoryboard* are all fairly self-explanatory. They cause the indicated *Storyboard* to pause, resume, stop, or skip to the end, as indicated by the *Action* name. The one property that all these *Action* classes have in common is the *BeginStoryboardName* property. This property indicates the name of the *BeginStoryboard* object that the action is to affect. The following example demonstrates a *StopStoryboard* action that stops the *BeginStoryBoard* object named *stb1:*

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
         <BeginStoryboard Name="stb1">
            <Storyboard>
               <DoubleAnimation Duration="0:0:5"
                  Storyboard.TargetProperty="Height" To="200" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger.Actions>
   </EventTrigger>
   <EventTrigger RoutedEvent="Button.MouseLeave">
```

```
      <EventTrigger.Actions>
         <StopStoryboard BeginStoryboardName="stb1" />
      </EventTrigger.Actions>
   </EventTrigger>
</Style.Triggers>
```

All *Actions* that affect a particular *Storyboard* object must be defined in the same *Triggers* collection. The previous example shows both of these triggers being defined in the *Button.Triggers* collection. If you were to define these triggers in separate *Triggers* collections, storyboard actions would not function.

The *SetStoryboardSpeedRatio* action sets the speed ratio for the entire *Storyboard* and all *Animation* objects in that *Storyboard.* In addition to *BeginStoryboardName,* you must set the *SpeedRatio* property of this *Action* as well. The following example demonstrates a *SetStoryboardSpeedRatio* action that speeds the referenced *Storyboard* by a factor of 2:

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
         <BeginStoryboard Name="stb1">
            <Storyboard>
               <DoubleAnimation Duration="0:0:5"
                  Storyboard.TargetProperty="Height" To="200" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger.Actions>
   </EventTrigger>
   <EventTrigger RoutedEvent="Button.MouseLeave">
      <EventTrigger.Actions>
         <SetStoryboardSpeedRatio BeginStoryboardName="stb1" SpeedRatio="2" />
      </EventTrigger.Actions>
   </EventTrigger>
</Style.Triggers>
```

The *SeekStoryboard* action requires two additional properties to be set. The *Origin* property can be either a value of *BeginTime* or of *Duration* and specifies how the *Offset* property is applied. An *Origin* value of *BeginTime* specifies that the *Offset* is relative to the beginning of the *Storyboard.* An *Origin* value of *Duration* specifies that the *Offset* is relative to the *Duration* property of the *Storyboard.* The *Offset* property determines the amount of the offset to jump to in the animation. The following example shows a *SeekStoryboard* action that skips the referenced timeline to 5 seconds ahead from its current point in the timeline.

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
```

```
         <BeginStoryboard Name="stb1">
            <Storyboard>
               <DoubleAnimation Duration="0:0:10"
                  Storyboard.TargetProperty="Height" To="200" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger.Actions>
   </EventTrigger>
   <EventTrigger RoutedEvent="Button.MouseLeave">
      <EventTrigger.Actions>
         <SeekStoryboard BeginStoryboardName="stb1" Origin="BeginTime"
            Offset="0:0:5" />
      </EventTrigger.Actions>
   </EventTrigger>
</Style.Triggers>
```

Using Property Triggers with *Animations*

In the examples shown in this section, you have seen *Actions* being hosted primarily in *EventTrigger* objects. You can also host *Action* objects in other kinds of *Triggers. Trigger, MultiTrigger, DataTrigger, and MultiDataTrigger* objects host two *Action* collections: *EnterActions* and *ExitActions* collections.

The *EnterActions* collection hosts a set of *Actions* that are executed when the *Trigger* is activated. Conversely, the *ExitActions* collection hosts a set of *Actions* that are executed when the *Trigger* is deactivated. The following demonstrates a *Trigger* that begins a *Storyboard* when activated and stops that *Storyboard* when deactivated:

```
<Trigger Property="IsMouseOver" Value="True">
   <Trigger.EnterActions>
      <BeginStoryboard Name="stb1">
         <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="FontSize"
               To="20" Duration="0:0:.5" />
         </Storyboard>
      </BeginStoryboard>
   </Trigger.EnterActions>
   <Trigger.ExitActions>
      <StopStoryboard BeginStoryboardName="stb1" />
   </Trigger.ExitActions>
</Trigger>
```

Managing the Playback Timeline

Both the *Animation* class and the *Storyboard* class contain several properties that allow you to manage the playback timeline with a fine level of control. Each of these properties is discussed in this section. When a property is set on an *Animation,* the setting affects only that animation. Setting a property on a *Storyboard,* however, affects all *Animation* objects it contains.

AccelerationRatio and *DecelerationRatio*

The *AccelerationRatio* and *DecelerationRatio* properties allow you to designate a part of the timeline for acceleration and deceleration of the animation speed, rather than starting and playing at a constant speed. This is used sometimes to give an animation a more "natural" appearance. These properties are expressed in fractions of 1 and represent a percentage value of the total timeline. Thus, an *AccelerationRatio* with a value of .2 indicates that 20 percent of the timeline should be spent accelerating to the top speed. So the *AccelerationRatio* and *DecelerationRatio* properties should be equal to or less than 1 when added together. This example shows an *Animation* with an *AccelerationRatio* of .2:

```
<DoubleAnimation Duration="0:0:5" AccelerationRatio="0.2"
   Storyboard.TargetProperty="Height" To="200" />
```

AutoReverse

As the name implies, the *AutoReverse* property determines whether the animation automatically plays out in reverse after the end is reached. A value of *True* indicates that the *Animation* will play in reverse after the end is reached. *False* is the default value. The following example demonstrates this property:

```
<DoubleAnimation Duration="0:0:5" AutoReverse="True"
   Storyboard.TargetProperty="Height" To="200" />
```

FillBehavior

The *FillBehavior* property determines how the *Animation* behaves after it has completed. A value of *HoldEnd* indicates that the *Animation* holds the final value after it has completed, whereas a value of *Stop* indicates that the *Animation* stops and returns to the beginning of the timeline when completed. An example is shown here:

```
<DoubleAnimation Duration="0:0:5" FillBehavior="Stop"
   Storyboard.TargetProperty="Height" To="200" />
```

The default value for *FillBehavior* is *HoldEnd*.

RepeatBehavior

The *RepeatBehavior* property determines if and how an animation repeats. The *RepeatBehavior* property can be set in three ways. First, it can be set to *Forever,* which indicates that an *Animation* repeats for the duration of the application. Second, it can be set to a number followed by the letter *x* (for example, *2x*), which indicates the number of times to repeat the animation. Third, it can be set to a *Duration,* which indicates the amount of time that an *Animation* plays, irrespective of the number of iterations. The following three examples demonstrate these settings. The first demonstrates an *Animation* that repeats forever, the second an *Animation* that repeats three times, and the third an *Animation* that repeats for 1 minute:

```
<DoubleAnimation Duration="0:0:5" RepeatBehavior="Forever"
   Storyboard.TargetProperty="Height" To="200" />
<DoubleAnimation Duration="0:0:5" RepeatBehavior="3x"
   Storyboard.TargetProperty="Height" To="200" />
<DoubleAnimation Duration="0:0:5" RepeatBehavior="0:1:0"
   Storyboard.TargetProperty="Height" To="200" />
```

SpeedRatio

The *SpeedRatio* property allows you to speed up or slow down the base timeline. The *SpeedRatio* value represents the coefficient for the speed of the *Animation*. Thus, an *Animation* with a *SpeedRatio* value of 0.5 takes twice the standard time to complete, whereas a value of 2 causes the *Animation* to complete twice as fast. An example is shown here:

```
<DoubleAnimation Duration="0:0:5" SpeedRatio="0.5"
   Storyboard.TargetProperty="Height" To="200" />
```

Animating Non-Double Types

Most of the examples that you have seen in this lesson have dealt with the *DoubleAnimation* class, but in fact a class exists for every animatable data type. For example, the *ColorAnimation* class allows you to animate a color change, as shown here:

```
<Button Height="23" Width="100" Name="Button1">
   <Button.Background>
      <SolidColorBrush x:Name="myBrush" />
   </Button.Background>
   <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
         <BeginStoryboard>
            <Storyboard>
               <ColorAnimation Storyboard.TargetName="myBrush"
                  Storyboard.TargetProperty="Color" From="Red" To="LimeGreen"
                  Duration="0:0:5" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger>
   </Button.Triggers>
</Button>
```

In this example, when the button is clicked, the background color of the button gradually changes from red to lime green over the course of 5 seconds.

NOTE In the standard Windows theme, this animation may conflict with other animations in the button's default template, so you might need to mouse out of the button and defocus it to see the full effect.

Animation with Key Frames

Up until now, all the animations you have seen have used linear interpolation—that is, the animated property changes take place over a linear timeline at a linear rate. You also can create nonlinear animations by using key frames.

Key frames are waypoints in an animation. Instead of allowing the *Animation* to progress linearly from beginning to end, key frames divide the animation up into short segments. The animation progresses from the beginning to the first key frame, then the next, and through the *KeyFrames* collection until the end of the animation is reached. Each key frame defines its own *Value* and *KeyTime* properties, which indicate the value that the *Animation* will represent when it reaches the key frame and the time in the *Animation* at which that frame will be reached.

Every data type that supports a linear *Animation* type also supports a key-frame *Animation* type, and some types that do not have linear animation types have key-frame *Animation* types. The key-frame *Animation* types are named *<TargetType>AnimationUsingKeyFrames,* where *<TargetType>* represents the name of the *Type* animated by the *Animation*. Key-frame *Animation* types do not support the *From, To,* and *By* properties; rather, the course of the *Animation* is defined by the collection of key frames.

There are three different kinds of key frames. The first is linear key frames, which are named *Linear<TargetType>KeyFrame*. These key frames provide points in an *Animation* that are interpolated between in a linear fashion. The following example demonstrates the use of linear key frames:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Height">
   <LinearDoubleKeyFrame Value="10" KeyTime="0:0:1" />
   <LinearDoubleKeyFrame Value="100" KeyTime="0:0:2" />
   <LinearDoubleKeyFrame Value="30" KeyTime="0:0:4"/>
</DoubleAnimationUsingKeyFrames>
```

In the preceding example, the *Height* property goes from its starting value to a value of 10 in the first second, then to a value of 100 in the next second, and finally returns to a value of 30 in the last 2 seconds. The progression between each segment is interpolated linearly. In this example, it is similar to having several successive linear *Animation* objects.

Discrete Key Frames

Some animatable data types do not support gradual transitions under any circumstances. For example, the *String* type can only accept discrete changes. You can use discrete key frame objects to make discrete changes in the value of an animated property. Discrete key frame classes are named *Discrete<TargetType>KeyFrame,* where *<TargetType>* is the *Type* being animated. Like linear key frames, discrete key frames use a *Value* and a *KeyTime* property to set the parameters of the key frame. The following example demonstrates an animation of a *String* using discrete key frames:

```
<StringAnimationUsingKeyFrames Storyboard.TargetProperty="Content">
   <DiscreteStringKeyFrame Value="Soup" KeyTime="0:0:0" />
   <DiscreteStringKeyFrame Value="Sous" KeyTime="0:0:1" />
   <DiscreteStringKeyFrame Value="Sots" KeyTime="0:0:2" />
   <DiscreteStringKeyFrame Value="Nots" KeyTime="0:0:3" />
   <DiscreteStringKeyFrame Value="Nuts" KeyTime="0:0:4" />
</StringAnimationUsingKeyFrames>
```

Spline Key Frames

Spline key frames allow you to define a Bézier curve that expresses the relationship between animation speed and animation time, thus allowing you to create animations that accelerate and decelerate in complex ways. While the mathematics of Bézier curves is beyond the scope of this lesson, a Bézier curve is simply a curve between two points whose shape is influenced by two control points. Using spline key frames, the start and end points of the curve are always (0,0) and (1,1) respectively, so you must define the two control points. The *KeySpline* property accepts two points to define the Bézier curve, as seen here:

```
<SplineDoubleKeyFrame Value="300" KeyTime="0:0:6" KeySpline="0.1,0.8 0.6,0.6" />
```

Spline key frames are difficult to create with the intended effect without complex design tools, and are most commonly used when specialized animation design tools are available.

Using Multiple Types of Key Frames in an Animation

You can use multiple types of key frames in a single animation—you can freely intermix *LinearKeyFrame*, *DiscreteKeyFrame*, and *SplineKeyFrame* objects in the *KeyFrames* collection. The only restriction is that all key frames you use must be appropriate to the *Type* that is being animated. *String* animations, for example, can use only *DiscreteStringKeyFrame* objects.

Quick Check

- What are the different types of key frame objects? When would you use each one?

Quick Check Answer

- There are *LinearKeyFrame*, *DiscreteKeyFrame*, and *SplineKeyFrame* objects. *LinearKeyFrame* objects indicate a linear transition from the preceding property value to the value represented in the key frame. *DiscreteKeyFrame* objects represent a sudden transition from the preceding property value to the value represented in the key frame. *SplineKeyFrame* objects represent a transition whose rate is defined by the sum of an associated Bézier curve. You would use each of these types when the kind of transition represented was the kind of transition that you wanted to incorporate into your user interface. In addition, some animation types can use only *DiscreteKeyFrames*.

Creating and Starting Animations in Code

All the *Animation* objects that you have seen so far in this lesson were created declaratively in XAML. However, you can create and execute *Animation* objects just as easily in code as well.

The process of creating an *Animation* should seem familiar to you; as with other .NET objects, you create a new instance of your *Animation* and set the relevant properties, as seen in this example:

```
' VB
Dim aAnimation As New System.Windows.Media.Animation.DoubleAnimation()
aAnimation.From = 20
aAnimation.To = 300
aAnimation.Duration = New Duration(New TimeSpan(0, 0, 5))
aAnimation.FillBehavior = Animation.FillBehavior.Stop

// C#
System.Windows.Media.Animation.DoubleAnimation aAnimation = new
   System.Windows.Media.Animation.DoubleAnimation();
aAnimation.From = 20;
aAnimation.To = 300;
aAnimation.Duration = new Duration(new TimeSpan(0, 0, 5));
aAnimation.FillBehavior = Animation.FillBehavior.Stop;
```

After the *Animation* has been created, however, the obvious question is: How do you start it? When creating *Animation* objects declaratively, you must use a *Storyboard* to organize your *Animation* and an *Action* to start it. In code, however, you can use a simple method call. All WPF controls expose a method called *BeginAnimation,* which allows you to specify a dependency property on that control and an *Animation* object to act on that dependency property. The following code shows an example:

```
' VB
Button1.BeginAnimation(Button.HeightProperty, aAnimation)

// C#
button1.BeginAnimation(Button.HeightProperty, aAnimation);
```

Lab: Improving Readability with Animations

In this lab, you improve upon your solution to the lab in Lesson 1 of this chapter. You remove the triggers that cause the *FontSize* to expand and instead use an *Animation* to make it look more natural. In addition, you create *Animation* objects to increase the size of the control when the mouse is over it.

Exercise: Animating High-Contrast Styles

1. Open the completed solution from the lab from Lesson 1 of this chapter.
2. In each of the *Styles,* remove the *FontSize Setter* that is defined in the *Trigger* and replace it with a *Trigger.EnterActions* and *Trigger.ExitActions* section, as shown here:

```
<Trigger.EnterActions>

</Trigger.EnterActions>
<Trigger.ExitActions>

</Trigger.ExitActions>
```

3. In each *Trigger.EnterActions* section, add a *BeginStoryboard* action, as shown here:

```
<BeginStoryboard Name="Storyboard1">

</BeginStoryboard>
```

4. Add the following *Storyboard* and *Animation* objects to the *BeginStoryboard* object in the style for the *TextBox.* Note that the values for the *ThicknessAnimation* object are crafted specifically for the completed version of the Lesson 1 lab on the CD. If you created your own solution, you need to recalculate these values:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize"
      To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,118,45,104" />
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

5. Add a similar *Storyboard* to the style for the *Label,* as shown here:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize" To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,62,46,-10" />
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

6. Add a similar *Storyboard* to the style for the *Button,* as shown here:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize" To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,0,46,52" />
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

7. Add the following line to the *Trigger.ExitActions* section of each *Style:*

```
<StopStoryboard BeginStoryboardName="Storyboard1" />
```

8. Press F5 to build and run your application. Now the *FontSize* expansion is animated and the control expands as well.

Lesson Summary

- Animation objects drive automated property changes over time. There are three different types of *Animation* objects—linear animations, key frame–based animations, and path-based animations. Every animatable type has at least one *Animation* type associated with it, and some types have more than one type of *Animation* that can be applied.
- Storyboard objects organize one or more *Animation* objects. Storyboard objects determine what objects and properties their contained *Animation* objects are applied to.
- Both *Animation* and *Storyboard* objects contain a variety of properties that control *Animation* playback behavior.
- Storyboard objects that are created declaratively are activated by a *BeginStoryboard* action in the *Actions* collection of a *Trigger. Triggers* also can define actions that pause, stop, and resume *Storyboard* objects, as well as performing other *Storyboard*-related functions.

- Key frame animations define a series of waypoints through which the *Animation* passes. There are three kinds of key frames: linear key frames, discrete key frames, and spline key frames. Some animatable types, such as *String,* support only discrete key frames.
- You can create and apply *Animation* objects in code. When doing this, you do not need to define a *Storyboard* object; rather, you call the *BeginAnimation* method on the element with which you want to associate the *Animation*.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Animations." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. How many times does the *Animation* shown here repeat (not counting the first iteration)?

```
<DoubleAnimation Duration="0:0:15" RepeatBehavior="0:1:0"
   Storyboard.TargetProperty="Height" To="200" />
```

   A. 0

   B. 1

   C. 2

   D. 3

2. Look at this *Animation:*

```
<DoubleAnimation Duration="0:0:5" From="30" By="80" To="200"
Storyboard.TargetProperty="Height" />
```

   Assuming that the element whose *Height* property it animates begins with a *Height* of 50, what is the value of the element after the animation has completed?

   A. 50

   B. 110

   C. 130

   D. 200

Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can do any or all of the following:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- *Styles* allow you to define consistent visual styles for your application by using a collection of *Setters*. They usually are defined as a *Resource* and referenced in XAML, though they can be set inline or dynamically. *Styles* can be inherited from other styles and applied to all instances of a particular type.
- Triggers respond to changes in the application environment. Property triggers and multi-triggers listen for changes in property values, and data triggers and multi-data-triggers listen for changes in bound values. When one of these triggers is activated, its *Setters* collection is applied. *EventTriggers* listen for a routed event and execute *Actions* in response to that event.
- Property values follow a strict order of precedence depending on how they are set.
- *Animation* objects drive automated property changes over time. There are three different types of *Animation* objects—linear animations, key frame–based animations, and path-based animations. Every animatable type has one or more *Animation* classes that can be used with it. *Animations* are organized by *Storyboard* objects, which are themselves controlled by *Action* objects that are activated in the *Action* collections of *Trigger* objects.
- *Animations* that use key frames provide waypoints that the *Animation* visits as it progresses. Key frames can be linear, spline-based, or discrete.
- You can create and apply *Animation* objects in code. When doing this, you do not need to define a *Storyboard* object, but rather you call the *BeginAnimation* method on the element with which you want to associate the *Animation*.

Key Terms

- Action
- Animation
- Key Frame
- Setter
- Storyboard
- Style
- Trigger

Case Scenarios

In the following case scenarios, you apply what you've learned about how to use controls to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Cup Fever

You've had a little free time around the office, and you decided to write a simple but snazzy application to organize and display results from World Cup soccer matches. The technical details are all complete: You've located a Web service that feeds up-to-date scores, and you've created a database that automatically applies updates from this service for match results and keeps track of upcoming matches. The database is exposed through a custom data object built on *ObservableCollection*<> lists. All that remains are the finishing touches. Specifically, when users choose an upcoming match from a drop-down box at the top of the window, you want the window's color scheme to match the colors of the teams in the selected matchup.

Technical Requirements

- The user interface is divided into two sections, each of which is built on a Grid container. Each section represents a team in the current or upcoming match. The user interface for each section must apply the appropriate team colors automatically when a new match is chosen.

Question

Answer the following question for all your office mates, who are eagerly awaiting the application's completion.

- How can you implement these color changes to the user interface?

Case Scenario 2: A Far-Out User Interface

Our friends with the questionable taste are back. They were so impressed with the work you did for them back in Chapter 4 that they've asked you to design a user interface that further pushes the envelope of good design sensibilities. Rather than having a static tie-dyed appearance, now they want the background to be a constantly changing multicolored experience. The idea of using a *RadialGradientBrush* to paint the background of the window is still acceptable, but they want the center of the gradient to change over time and they want the colors of the background to change.

Question

Answer the following question for your manager:

- How can we implement this appearance?

Suggested Practices

- Create an *Animation* that moves elements across the user interface. Alternatively, use linear animations and key frame animations to explore a variety of different animation styles. Animate other properties of UI elements as well, such as the color, size, and content.
- Use *Animations* to create a slideshow application that reads all the image files in a given directory and displays each image for 10 seconds before automatically switching to the next one. Note that you have to create and apply the *Animation* in code.
- Modify the solution from Lesson 2 of Chapter 6, "Converting and Validating Data," to create styles for the application that includes *DataTriggers* that automatically apply styles based on the *CompanyName* of the selected record.
- Modify the solution from the second lab in this chapter to reverse the *Animation* instead of stopping it when the mouse exits the control.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-502 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO Practice tests

For details about all the practice test options available, see the section "How to Use the Practice Tests," in this book's Introduction.